Titre: Title:	Advancing Software Instrumentation Through Cost Optimization
Auteur: Author:	Amir Haghshenas
Date:	2026
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Haghshenas, A. (2026). Advancing Software Instrumentation Through Cost Optimization [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/72273/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Advancing Software Instrumentation Through Cost Optimization

AMIR HAGHSHENAS
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Janvier 2026

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Advancing Software Instrumentation Through Cost Optimization

présentée par **Amir HAGHSHENAS**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Gilles PESANT, président
Michel DAGENAIS, membre et directeur de recherche
Naser EZZATI-JIVAN, membre et codirecteur de recherche
Heng LI, membre et codirecteur de recherche
Guy BOIS, membre
Ferhat KHENDEK, membre externe

DEDICATION

To my beloved wife, whose unwavering love, patience, and encouragement sustained me throughout this academic journey, vous me manquerez...

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisors, Professor Michel Dagenais and Professor Naser Ezzati-Jivan, for their exceptional guidance, patience, and unwavering support throughout my PhD journey. Their mentorship has been instrumental in shaping my research vision and academic development. Professor Dagenais's profound insights and understanding have taught me invaluable lessons that extend beyond academia, helping me maintain perspective and balance in both my professional and personal life. Professor Ezzati-Jivan's expertise and thoughtful guidance have been crucial in navigating the complexities of my research. I am truly grateful for the opportunity to work under their supervision.

I extend my sincere thanks to Professor Heng Li, my co-supervisor, whose valuable input and fresh perspectives have significantly enriched my work and broadened my understanding of the field.

Special thanks to my parents and family for their unconditional love, endless support, and countless sacrifices throughout this journey. Your belief in me has been my constant source of strength, and words cannot adequately express my gratitude for everything you have done.

To my research colleagues and labmates—thank you for creating such a collaborative and supportive environment. Your insights, discussions, and shared experiences have made this journey not only productive but also enjoyable.

To my friends, thank you for your understanding, encouragement, and for helping me maintain balance during the most challenging times. Your presence has made all the difference through the ups and downs of this journey.

I would like to thank the members of my examination committee for accepting to be part of my jury and for their valuable feedback and constructive comments that have helped to strengthen this work.

Finally, I gratefully acknowledge the financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC), as well as our industry partners for funding this research project and making this work possible.

RÉSUMÉ

L'analyse de performance logicielle est devenue critique pour assurer la fiabilité des systèmes et optimiser l'utilisation des ressources dans les environnements informatiques modernes. À mesure que les architectures logicielles ont évolué des systèmes monolithiques vers des microservices distribués servant des milliards d'utilisateurs, le défi de maintenir une visibilité complète des performances tout en opérant sous des contraintes de coût strictes s'est intensifié. Les approches traditionnelles d'instrumentation logicielle—principalement à travers le traçage d'exécution et la journalisation—introduisent une surcharge significative qui s'étend au-delà des simples pénalités de temps d'exécution pour englober la consommation mémoire, les exigences de stockage, l'utilisation de la bande passante réseau et le fardeau de maintenance. Le défi fondamental réside dans l'atteinte d'un équilibre optimal entre la couverture de surveillance et ces coûts associés, car une instrumentation excessive peut générer des volumes de données massifs et une dégradation sévère des performances, tandis qu'une surveillance insuffisante crée des angles morts critiques dans l'observabilité du système.

La majorité de la recherche en instrumentation logicielle s'est concentrée sur des questions fondamentales de où instrumenter, quelles informations capturer, ou comment implémenter les mécanismes de surveillance. Cependant, seul un nombre limité d'études ont abordé le défi critique d'équilibrer l'instrumentation pour obtenir des données diagnostiques de haute qualité tout en minimisant les coûts associés. Cette lacune laisse le compromis essentiel entre l'exhaustivité de la surveillance et les coûts d'instrumentation largement non adressé par des approches systématiques. Par conséquent, les développeurs doivent s'appuyer sur des processus de prise de décision ad hoc pour équilibrer ces exigences concurrentes, résultant souvent en des configurations d'instrumentation sous-optimales qui soit compromettent les performances du système par des coûts excessifs ou échouent à fournir une observabilité adéquate pour le débogage et l'optimisation.

Cette thèse aborde le défi fondamental du placement systématique d'instrumentation dans la surveillance de performance logicielle à travers un cadre exhaustif conscient des coûts. Le problème traité était le manque d'approches systématiques pour équilibrer l'efficacité de surveillance contre plusieurs contraintes de coût simultanément. Une taxonomie de coûts multidimensionnelle englobant douze dimensions de coût distinctes a été développée, ainsi qu'un cadre basé sur l'optimisation qui identifie automatiquement des configurations d'instrumentation quasi-optimales. Le cadre formule le placement d'instrumentation comme un problème d'optimisation de sac à dos, permettant aux praticiens de spécifier des objectifs

de surveillance tout en respectant les contraintes de coût. L'évaluation sur trois applications diverses—un benchmark SPEC CPU, une base de données Redis, et un serveur web Apache HTTPD—a démontré que les fonctions sélectionnées se classaient constamment parmi les premières 12-27% basé sur des mesures réelles de variabilité du temps d'exécution. La deuxième contribution aborde les défis de scalabilité de l'analyse de données de trace dans les systèmes de microservices à grande échelle. Le problème était que la surveillance exhaustive génère des volumes de données prohibitifs qui submergent les capacités de stockage et d'analyse. Une approche d'apprentissage automatique en deux phases utilisant des algorithmes de gradient boosting a été développée pour identifier les caractéristiques les plus critiques pour la prédiction d'utilisation CPU et mémoire. En utilisant un ensemble de données exhaustif du cluster de production d'Alibaba couvrant plus de 40 000 nœuds et 470 000 conteneurs, l'approche a atteint plus de 69% de réduction du volume de données de trace tout en maintenant ou améliorant la précision de modélisation, avec des scores RMSE de 0,02 pour le CPU et 0,13 pour l'utilisation mémoire comparé à 0,08 et 0,14 avec l'ensemble de données complet.

La troisième contribution fournit des fondations empiriques pour comprendre la relation entre les pratiques de journalisation et l'efficacité de résolution de bogues à travers différents domaines logiciels. Le problème abordé était la compréhension limitée de si les pratiques actuelles de journalisation améliorent réellement les résultats de débogage et comment les perceptions des développeurs s'alignent avec les comportements réels. Une étude de méthodes mixtes analysant plus de 572 000 fonctions à travers dix projets Java majeurs a été menée, combinée avec des sondages de 58 développeurs professionnels. La recherche a révélé des variations significatives dépendantes du domaine dans l'efficacité de la journalisation, avec les projets d'infrastructure montrant de fortes associations statistiques entre la journalisation et l'implication de bogues (ratios de cotes 1,21-1,48) tandis que les projets de framework ont démontré des associations minimales (ratios de cotes près de 1,0). De plus, un écart substantiel a été identifié entre les perceptions des développeurs—74% des développeurs industriels prétendent des approches de journalisation équilibrées—et les patrons prédominamment réactifs observés dans l'analyse de code. Ce travail établit des bases et des patrons empiriques essentiels qui fournissent la fondation pour la recherche future pour investiguer les relations causales entre les pratiques de journalisation et l'efficacité de débogage, permettant des études qui peuvent suivre des sessions de débogage réelles et mesurer l'impact direct de différentes stratégies de journalisation sur les résultats de résolution de problèmes.

ABSTRACT

Software performance analysis has become critical for ensuring system reliability and optimizing resource utilization in modern computing environments. As software architectures have evolved from monolithic systems to distributed microservices serving billions of users, the challenge of maintaining comprehensive performance visibility while operating under strict cost constraints has intensified. Traditional approaches to software instrumentation, primarily through execution tracing and logging, introduce significant overhead that extends beyond simple execution time penalties to include memory consumption, storage requirements, network bandwidth utilization, and maintenance burden. The fundamental challenge lies in achieving optimal balance between monitoring coverage and these associated costs, as excessive instrumentation can generate massive data volumes and severe performance degradation, while insufficient monitoring creates critical blind spots in system observability.

Most of the software instrumentation research has focused on addressing fundamental questions of where to instrument, what information to capture, or how to implement monitoring mechanisms. However, only a limited number of studies have addressed the critical challenge of balancing instrumentation to achieve high-quality diagnostic data while minimizing associated costs. This gap leaves the essential trade-off between monitoring comprehensiveness and instrumentation costs largely unaddressed by systematic approaches. Consequently, developers must rely on ad-hoc decision-making processes to balance these competing requirements, often resulting in suboptimal instrumentation configurations that either compromise system performance through excessive costs or fail to provide adequate observability for debugging and optimization.

This thesis addresses the fundamental challenge of systematic instrumentation placement in software performance monitoring through a comprehensive cost-aware framework. The problem tackled was the lack of systematic approaches to balance monitoring effectiveness against multiple cost constraints simultaneously. A multi-dimensional cost taxonomy encompassing twelve distinct cost dimensions was developed, along with an optimization-based framework that automatically identifies near-optimal instrumentation configurations. The framework formulates instrumentation placement as a knapsack optimization problem, enabling practitioners to specify monitoring objectives while respecting cost constraints. Evaluation in three diverse applications, a SPEC CPU benchmark, Redis database, and Apache HTTPD web server, demonstrated that selected functions consistently ranked within the top 12-27% based on actual execution time variability measurements. Functions with high execution

time variability indicate performance bottlenecks, resource contention, or state-dependent behavior, making them critical monitoring targets for effective performance diagnosis.

The second contribution addresses the scalability challenges of trace data analysis in large-scale microservice systems. The problem was that comprehensive monitoring generates prohibitively large data volumes that overwhelm storage and analysis capabilities. A two-phase machine learning approach using gradient boosting algorithms was developed to identify the most critical features for CPU and memory utilization prediction. Using a comprehensive dataset from Alibaba's production cluster spanning 40,000+ nodes and 470,000+ containers, the approach achieved over 69% reduction in trace data volume while maintaining or improving modeling accuracy, with RMSE scores of 0.02 for CPU and 0.13 for memory utilization compared to 0.08 and 0.14 with the full dataset.

The third contribution provides empirical foundations for understanding the relationship between logging practices and bug resolution effectiveness across different software domains. The problem addressed was the limited understanding of whether current logging practices actually improve debugging outcomes and how developer perceptions align with actual behaviors. A mixed-methods study analyzing over 572,000 functions across ten major Java projects was conducted, combined with surveys of 58 professional developers. The research revealed significant domain-dependent variations in logging effectiveness, with infrastructure projects showing strong statistical associations between logging and bug involvement (odds ratios 1.21-1.48) while framework projects demonstrated minimal associations (odds ratios near 1.0). Additionally, a substantial gap was identified between developer perceptions—74% of industry developers claim balanced logging approaches— while actually, predominantly reactive patterns are observed in code analysis. This work establishes essential empirical baselines and patterns that provide the foundation for future research to investigate causal relationships between logging practices and debugging effectiveness, enabling studies that can track actual debugging sessions and measure the direct impact of different logging strategies on problem resolution outcomes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

IETF	Internet Engineering Task Force
OSI	Open Systems Interconnection

CHAPTER 1 INTRODUCTION

Software performance analysis is a cornerstone of modern software engineering, critical for ensuring system reliability, optimizing resource utilization, and sustaining high quality of service in today's complex computational environments. As software architectures have evolved from monolithic implementations to distributed microservices supporting billions of users, the challenge of understanding, predicting, and optimizing performance behavior has become a vital operational concern. Modern performance analysis focuses not just on overall system behavior, but specifically on diagnosing and improving software performance under diverse and dynamic workloads. The economic significance of this domain is staggering: debugging costs the global software industry approximately $312 billion annually, with developers spending roughly 50% of their programming time finding and fixing bugs [1]. Concurrently, the global log management market is projected to grow from $2.3 billion in 2021 to $7.0 billion by 2030, reflecting the recognition of the industry that effective monitoring capabilities are essential for maintaining software quality and operational reliability [2].

Advancements in performance analysis have shifted the field from traditional analytical models to sophisticated frameworks, often based on machine learning, that emphasize cost awareness, automation, and scalability. The central challenge facing current research and practice is maintaining comprehensive performance visibility while minimizing the overhead and cost of monitoring, especially in hyperscale deployments, where even minor performance degradations can escalate into significant disruptions or losses.

Addressing this challenge fundamentally depends on software instrumentation, primarily realized through execution tracing and logging. Tracing records detailed, fine-grained execution events such as function calls and resource accesses, enabling precise bottleneck identification and in-depth system analysis. In contrast, logging uses selective statements embedded by developers to capture key events or states, embedding semantic and domain-specific information that is indispensable for debugging and runtime understanding. As modern systems generate massive volumes of log data, the need for log management platforms—capable of efficiently collecting, indexing, and analyzing logs at scale—has become essential to support operational reliability and rapid incident response.

Both approaches have proven invaluable for performance analysis, yet each introduces distinct challenges and costs. The instrumentation required for comprehensive monitoring—whether

[1] https://www.jbs.cam.ac.uk/2013/research-by-cambridge-mbas-for-tech-firm-undo-finds-software-bugs-cost-the-industry-316-billion-a-year/

[2] https://www.businessresearchinsights.com/market-reports/log-management-software-market-111895

through tracing or logging—fundamentally alters system behavior and resource consumption patterns. These alterations manifest as various forms of overhead that extend far beyond simple execution time penalties, encompassing memory consumption, storage requirements, network bandwidth utilization, and maintenance burden [1].

The fundamental challenge in software instrumentation lies in achieving optimal balance between monitoring coverage and associated costs. Excessive instrumentation presents numerous problems that can undermine the very systems it aims to monitor. The performance overhead of comprehensive tracing can vary widely, from minimal impact in lightly instrumented cases to severe degradation in heavily instrumented systems. [2]. Beyond execution time impacts, excessive instrumentation generates massive data volumes—petabytes annually [3] for large systems—creating storage, transmission, and analysis challenges that can overwhelm organizational resources. The data deluge problem extends beyond mere volume: cluttered datasets filled with redundant or irrelevant information significantly increase the time and computational resources required to identify and resolve actual problems. Analysis overhead can reach 5,002.9x in extreme cases, with 99.87% of generated monitors being ultimately wasted, highlighting the inefficiency of indiscriminate instrumentation approaches [4].

Conversely, insufficient instrumentation creates equally problematic blind spots in system observability. Critical performance issues may go undetected until they manifest as user-facing failures, while root cause analysis becomes nearly impossible without adequate diagnostic information. The 2021 Facebook global outage [3] and the 2022 Atlassian mass site deletion incident [4] exemplify how sophisticated monitoring systems can miss catastrophic errors when instrumentation fails to capture the right information at critical points. These incidents, despite occurring in heavily monitored environments, demonstrate that the quantity of instrumentation alone does not guarantee effective observability—the quality and strategic placement of instrumentation points prove equally crucial.

The contrast between over-instrumentation and under-instrumentation underscores the critical need for systematic, cost-aware approaches to software monitoring. Traditional ad-hoc instrumentation decisions, where developers rely on intuition or experience to determine monitoring points, prove inadequate for modern software systems characterized by massive scale, complex interactions, and stringent performance requirements. The challenge intensifies in resource-constrained environments such as embedded systems, mobile applications, and real-time systems, where instrumentation overhead directly impacts core functionality. High-frequency trading systems cannot tolerate additional microsecond latencies, mobile ap-

[3] https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/

[4] https://www.atlassian.com/blog/atlassian-engineering/post-incident-review-april-2022-outage

plications must balance monitoring against battery consumption, and IoT devices operate within severe memory and bandwidth constraints.

This instrumentation challenge becomes particularly acute in two scenarios that increasingly characterize modern software development. First, large software systems with thousands or tens of thousands of functions present a high number of potential instrumentation sets, making manual selection impractical and error-prone. Second, multi-tenant cloud environments and microservice architectures introduce competing resource demands and complex interactions that traditional monitoring approaches fail to capture efficiently. The need for systematic approaches that can automatically balance monitoring effectiveness against multiple cost constraints while adapting to diverse operational requirements has become paramount.

1.1 Research Objectives

This thesis addresses these fundamental challenges through three interconnected research contributions that advance the state of cost-aware software instrumentation and analysis. Each contribution targets a specific aspect of the instrumentation cost problem while collectively establishing a comprehensive framework for systematic performance monitoring decisions.

More specifically, this thesis answers three main research questions in the domain of cost-aware performance analysis and software instrumentation, each representing a critical challenge that has hindered the adoption of comprehensive monitoring in production systems.

Research Question 1: How can the fundamental trade-offs between monitoring coverage and system overhead be systematically managed in software instrumentation decisions?

Our initial work, presented in "Balancing Costs and Insights: A Framework for Cost-Aware Software Instrumentation," establishes the theoretical and practical foundations for systematic instrumentation decisions. Through an extensive literature survey examining papers from 2000-2024, we identified and categorized twelve distinct cost dimensions associated with software instrumentation, ranging from immediate execution overhead to deferred maintenance burden. This multi-dimensional cost taxonomy represents the first systematic categorization that considers factors beyond execution time alone, examining costs across intrinsic versus extrinsic, operational versus non-operational, and immediate versus deferred dimensions.

Building upon this cost understanding, we developed an optimization-based framework that automatically identifies near-optimal instrumentation configurations for specified monitoring objectives while respecting cost constraints. The framework formulates instrumentation placement as a knapsack optimization problem [5], enabling practitioners to precisely determine instrumentation requirements while automatically selecting optimal code sections

based on multiple cost constraints simultaneously. Our prototype implementation, evaluated on three diverse applications—a SPEC CPU benchmark, Redis database, and Apache HTTPD web server—demonstrates that the framework effectively identifies high-variability functions that warrant monitoring attention, with selected functions consistently ranking within the top 12-27% based on actual execution time variability measurements.

Research Question 2: How can the scalability challenges of comprehensive system monitoring be addressed when trace data volumes become prohibitively large for analysis and storage?

Our second work addresses a critical component of the instrumentation cost spectrum: the analysis cost associated with processing massive trace data volumes. In "Automatic Reduction of Execution Trace Data Volume Using Gradient Boosting in Large-Scale Microservice Systems," we tackle the challenge of maintaining effective performance modeling capabilities while dramatically reducing the data collection and analysis burden. Using a comprehensive dataset from Alibaba's production cluster—spanning 13 days across 40,000+ bare-metal nodes and 470,000+ containers—we developed a two-phase machine learning approach that identifies the most critical features for accurate CPU and memory utilization prediction.

Through systematic application of gradient boosting algorithms, we demonstrated that performance modeling accuracy can be maintained or even improved while reducing trace data volume by more than 69%. The analysis revealed that the most influential features for resource utilization modeling are closely tied to inter-service communication, memory access, and database interactions—validating that targeted data collection focusing on critical architectural aspects can substantially decrease monitoring overhead without sacrificing analytical capability. This work provides empirical evidence that intelligent feature selection can address both the immediate costs of data collection and the deferred costs of data storage and analysis, contributing directly to more sustainable monitoring practices in large-scale systems.

Research Question 3: What is the relationship between logging practices and bug resolution effectiveness across different software domains, and how do developer perceptions align with actual logging behaviors?

Our third contribution establishes critical empirical foundations for understanding the actual effectiveness of current logging practices in software debugging contexts. In "Logging Practices in Software Bug Resolution: An Empirical Study," we conducted a comprehensive mixed-methods investigation analyzing over 572,000 functions across ten major open-source Java projects, combined with survey responses from 58 professional developers. This study provides the first large-scale empirical evidence of how logging patterns relate to bug reso-

lution activities, revealing significant domain-dependent variations and challenging common assumptions about logging effectiveness.

Our analysis uncovered that logging associations with bug involvement vary substantially across software domains, with infrastructure projects showing strong statistical associations (odds ratios 1.21-1.48) while framework projects demonstrate minimal associations (odds ratios near 1.0). Furthermore, we identified a significant gap between developer perceptions and actual practices: while 74% of industry developers claim to follow balanced logging approaches, code analysis reveals predominantly reactive patterns where developers add logging 1.2-3.8 times more frequently during bug fixes than regular development. These findings establish baseline patterns for understanding when and how logging actually contributes to debugging effectiveness, providing essential empirical grounding for cost-aware instrumentation decisions.

1.2 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 presents our literature review that covers the current state of research in performance analysis, software instrumentation practices, and the various costs associated with performance monitoring. Chapter 3 provides an overview of the three research papers that form the core contributions of this thesis. Chapter 4 presents the first article, which develops a systematic framework for cost-aware software instrumentation by establishing a multi-dimensional cost taxonomy and formulating instrumentation placement as an optimization problem that automatically identifies near-optimal configurations while respecting multiple cost constraints. Chapter 5 presents the second article, which addresses the challenges of scalability of trace data by developing a machine learning approach using gradient enhancement algorithms that reduces trace data volume while maintaining or improving the accuracy of performance modeling in large-scale microservice systems. Chapter 6 presents the third article, which provides empirical foundations for understanding the relationship between logging practices and bug resolution through mixed-methods analysis across ten Java projects. Finally, Chapter 7 concludes the thesis by synthesizing the contributions of all three research papers, discussing the limitations encountered in each study, and identifying future research directions that build upon the foundations established in this work to advance cost-aware performance analysis toward more systematic and evidence-based approaches.

1.3 Publications

1. **Haghshenas, Amir**, Nasser Ezzati-Jivan, and Michel Dagenais. "Automatic Reduction of Execution Trace Data Volume Using Gradient Boosting in Large-Scale Microservice Systems." 37th Canadian Conference on Artificial Intelligence (2024) - Accepted

2. **Haghshenas, Amir**, Nasser Ezzati-Jivan, and Michel Dagenais. "Balancing Costs and Insights: A Framework for Cost-Aware Software Instrumentation" - Journal of Software: Evolution and Process, 2025, (Under review)

3. **Haghshenas, Amir**, Nasser Ezzati-Jivan, and Michel Dagenais."Logging Practices in Software Bug Resolution: An Empirical Study" - Empirical Software Engineering, 2025 (Under Review)

CHAPTER 2 LITERATURE REVIEW

This literature review systematically examines the current state of research in software performance analysis, with particular emphasis on instrumentation practices and their associated costs. The review is organized into four complementary sections that collectively establish the theoretical foundations, practical challenges, and empirical evidence necessary to understand the complex landscape of cost-aware software monitoring. Our categorization follows a logical progression from broad performance analysis concepts to specific instrumentation implementation details, ultimately focusing on the cost dimensions that represent the central challenge addressed by this thesis.

The first section, Performance Analysis, establishes the broader context by examining three fundamental areas: performance modeling techniques for system behavior prediction, performance measurement frameworks for runtime data collection, and performance diagnosis methods for identifying and resolving performance issues. The second section, Software Instrumentation Practice, synthesizes research on practical implementation challenges, organizing the literature around three critical aspects: usability concerns affecting developer adoption, diagnosability requirements for effective problem resolution, and automation advances reducing manual instrumentation burden. The third section, Monitoring and Analysis Approaches in Software Systems, provides essential conceptual foundations by clarifying key terms and distinguishing between related but distinct approaches including tracing, profiling, debugging, and system monitoring. The final section, Instrumentation Costs, presents a comprehensive analysis of the twelve distinct cost dimensions identified through systematic literature review, spanning immediate execution overhead to long-term organizational impacts. This organizational structure enables a thorough understanding of how cost considerations permeate every aspect of software instrumentation, from theoretical foundations through practical implementation challenges.

2.1 Performance Analysis

Performance analysis encompasses the study of software system behavior under various operational conditions, with the goal of understanding, predicting, and optimizing computational performance. The field has evolved significantly in recent years, transitioning from traditional analytical approaches to sophisticated machine learning-driven frameworks that emphasize cost-awareness, automation, and practical scalability. Current research addresses the fundamental challenge of maintaining comprehensive performance visibility while operating under

strict cost and overhead constraints, particularly relevant for hyper-scale deployments serving billions of users across millions of servers.

This section examines three primary areas within performance analysis: performance modeling techniques that predict system behavior, performance measurement frameworks that collect runtime data, and performance diagnosis methods that identify and resolve performance issues.

2.1.1 Performance Modeling

Performance modeling involves the creation of mathematical or computational representations that predict system behavior under various conditions. Recent advances in this area have focused on developing generalizable models that can adapt to diverse hardware architectures and software configurations.

Li et al. [6] introduced PerfVec, a significant breakthrough in performance modeling that achieves generalizable program and architecture representations. Their approach demonstrates 15x speedup over existing techniques while maintaining cross-platform applicability, representing a paradigm shift from application-specific models to foundation model approaches that learn reusable representations across different hardware architectures and software stacks.

Machine learning approaches have become increasingly dominant in performance prediction, with deep learning frameworks specifically addressing the complexity of modern distributed systems. Research presented at SIGMETRICS 2025 demonstrates sophisticated applications including diffusion-based generative system surrogates for scalable optimization in virtual environments [7], and NetJIT's integration of traffic prediction with distributed ML system performance optimization [8].

Serverless ML performance modeling has emerged as a critical area due to the sensitivity of resource efficiency and SLO attainment to function design. Hui et al. [9] introduce predictive models for adaptive function granularity, utilizing linear regression, random forests, and deep reinforcement learning to dynamically optimize scheduling and resource allocation.

Advanced techniques for workload characterization and performance variation detection have incorporated machine learning methods to account for interference in multi-tenant environments. Alioth et al. [10] address multi-tenant interference prediction, providing a representative example of this direction.

2.1.2 Performance Measurement

Performance measurement encompasses the collection and analysis of runtime data to understand system behavior and identify performance characteristics. Recent research has focused on developing low-overhead measurement techniques capable of operating at hyperscale while maintaining high precision and accuracy.

A turning point in production performance monitoring is represented by FBDetect [11], which demonstrates the ability to detect performance regressions as small as 0.005% across 800,000 monitored time series at Meta's hyperscale deployment. This system employs advanced statistical filtering, fleet-wide stack trace sampling, and sophisticated false positive reduction techniques, reportedly saving approximately 4,000 servers annually through early regression detection.

Architecture-aware monitoring techniques have achieved significant overhead reduction while maintaining measurement quality. Recent work on GPU-optimized profiling achieves 96.16% accuracy with only 4.61x overhead through innovative call path memoization mechanisms [12]. Similarly, GPU-FPX demonstrates 16x performance improvements over comparable tools through data-parallel floating-point exception detection optimized for massively parallel architectures [13].

Network and distributed system monitoring has achieved practical breakthrough results through innovative approaches. INT-MC's matrix completion approach reduces in-band network telemetry overhead while maintaining comprehensive visibility [14], and VTrace provides automatic diagnostic capabilities for cloud-scale overlay networks through selective packet tracking approaches [15].

2.1.3 Performance Diagnosis

Performance diagnosis involves identifying the root causes of performance problems and providing actionable insights for system optimization. This area has evolved significantly through the integration of machine learning techniques with traditional causal reasoning approaches.

Distributed tracing has evolved beyond simple request tracking to intelligent instrumentation control systems. Astraea [16] represents a major advance in probabilistic tracing, using Bayesian learning and multi-armed bandits to achieve 92% performance variation localization using only 25% of instrumentation overhead. This online learning approach reduces false positives by 50% while accelerating training by 28x compared to traditional sampling methods.

Critical path analysis has proven essential for microservice diagnosis at scale. It focuses on

identifying the longest latency-contributing path in request execution graphs, thereby pinpointing the components most responsible for end-to-end performance degradation. CRISP's deployment at Uber [17] demonstrates practical impact across 40K endpoints serving 100M+ users, identifying 5 critical performance issues and reducing anomaly detection false positives by 50%. The DAG-based approach provides both top-down and bottom-up analysis capabilities essential for complex service mesh architectures.

Recent studies have proposed workflow-centric techniques [18], [19], [3]. These techniques capture the workflow of causally related events (e.g., work done to process a request) among the services of a distributed system, as well as their performance metrics and traced information (e.g., RPCs execution times, HTTP headers, resource consumption, execution nodes or application logs). Although these techniques can help collect workflow information on causally related performance data, there is still a gap on how to use workflow-centric traces to provide suggestions during the performance analysis. Popular workflow-centric technologies such as Zipkin [1] and Jaeger [2] provide Gantt charts to debug performance issues related to RPC degradation. Gantt charts are used to show individual requests. However, the effectiveness of Gantt charts is restricted to cases where the targets of the analysis are one or few requests. Indeed, as the number of requests under analysis grows, the causes of performance degradations potentially increase.

Canopy [18] processes real-time data and extracts user-specific features. The output of this method is a database aggregating a large number of traces. Pivot tracing [20] provides users with the ability to define tracing metrics at run-time, even when crossing component or machine boundaries. Other studies also have taken place in the field of supporting performance diagnosis. Sambavisan *et al.* [19] performed a comparison between three well-known visualization approaches in the context of presenting the results of automated performance root cause analysis approach. Another study by Malik [21] used performance counter data of a load test to craft performance signatures, and use them to pinpoint the subsystems responsible for the performance violations. They presented four machine learning approaches to help performance analysts to compare load tests more effectively, in order to detect performance deviations which may lead to service level agreement (SLA) violations, and to provide them with a smaller and manageable set of important performance counters.

Integration challenges represent the primary research frontier in performance diagnosis. Current research typically address individual aspects of performance problems rather than provid-

[1] https://zipkin.io/
[2] https://www.jaegertracing.io/

ing comprehensive frameworks that integrate prediction, detection, and remediation. Multi-modal analysis combining logs, metrics, traces, and topology remains an active research area with significant opportunities for advancement.

2.2 Monitoring and Analysis Approaches in Software Systems

This section establishes the fundamental concepts and terminology used throughout this thesis. Understanding these definitions is essential for comprehending the methodologies, frameworks, and analyses presented in subsequent chapters. The terminology spans multiple domains including software tracing, performance analysis, debugging practices, and system monitoring, drawing from recent advances in premier research venues.

2.2.1 Instrumentation

Software instrumentation represents the systematic addition of monitoring code to applications for collecting runtime behavior information. Instrumentation approaches span multiple implementation strategies including static code modification, dynamic binary instrumentation, and compiler-based automatic instrumentation. Each approach involves distinct trade-offs between analytical capabilities and implementation complexity.

Cost-aware instrumentation addresses the fundamental challenge of balancing monitoring effectiveness against the overhead introduced by data collection activities [22]. This approach recognizes that comprehensive monitoring introduces costs spanning execution time overhead, memory consumption, storage requirements, and maintenance burden. Systematic cost-aware frameworks enable optimization of instrumentation placement to maximize analytical value while respecting operational constraints.

Static instrumentation involves modifying source code to include monitoring statements before compilation, providing precise control over data collection activities but requiring access to application source code [23]. Dynamic instrumentation applies monitoring capabilities to compiled applications without source code modification, offering deployment flexibility at the cost of reduced optimization opportunities.

Recent advances in dynamic analysis have achieved remarkable overhead reductions through selective instrumentation approaches. Modern dynamic analysis frameworks now achieve only 9% runtime overhead compared to traditional approaches' 5x-100x overhead through intelligent selection algorithms that reduce traced instructions from 70% to 11% [24]. Binary instrumentation frameworks have evolved to combine the safety of traditional approaches with the performance of static transformation, achieving 2% overhead while maintaining safety

through sophisticated error handling mechanisms [25]. Dynamic analysis frameworks for specific programming languages have matured significantly, with Python frameworks achieving 1.2x-16x runtime overhead while being 5.6%-88.6% faster than built-in tracing APIs through hierarchical analysis hooks [26]. These advances demonstrate that comprehensive dynamic analysis is becoming viable for production environments where overhead constraints previously prohibited such approaches.

Tracing

Tracing represents a systematic method for understanding software system behavior by collecting detailed information about program execution paths and runtime events [27]. A tracer is the software component responsible for implementing this data collection process, enabling comprehensive visibility into system behavior during execution. Unlike traditional logging approaches that capture high-level application events, tracing typically records millions of low-level events at high frequencies, providing detailed insights into execution flows and system interactions [28].

Tracing encompasses both kernel-space and user-space data collection approaches, each serving distinct analytical purposes. User-space tracing focuses on application-level events and requires instrumentation of source code through either static instrumentation (applied before compilation) or dynamic instrumentation (applied to binary executables at runtime). This approach provides detailed insights into application logic and business workflows but requires access to application source code and may necessitate recompilation for static approaches.

Kernel-space tracing operates at the operating system level, capturing system-wide events including thread scheduling, file system operations, network activity, and system call patterns [29]. This approach operates independently of application code modifications, enabling comprehensive system analysis without impacting application development cycles. Kernel-space tracing proves particularly valuable for detecting system-level performance bottlenecks and resource contention that may not be visible through application-level monitoring alone.

Distributed tracing extends these concepts to microservice architectures, where request flows span multiple services and network boundaries. Recent research has demonstrated that distributed tracing can be achieved through network-level trace reconstruction, eliminating traditional application instrumentation requirements while maintaining comprehensive observability [30]. The implementation of tracing inevitably introduces execution overhead, representing the additional computational cost required to collect and process trace data, with recent studies showing throughput reductions of 19-80% for microservices and latency increases up to 175% for serverless applications [31].

Tracing Tools and Frameworks

Contemporary software systems employ diverse tracing tools, each optimized for specific use cases and operational environments. LTTng (Linux Trace Toolkit next generation) provides low-overhead, high-performance tracing capabilities that enable integrated analysis of both kernel-space and user-space events simultaneously [32]. LTTng utilizes approximately 2% of CPU time under heavy workloads and supports both static tracepoints and dynamic instrumentation through Linux Kernel Markers and Kprobes.

eBPF (extended Berkeley Packet Filter) represents a significant advancement in kernel-level tracing, providing a virtual machine framework for executing custom code within the Linux kernel without requiring kernel modifications [33]. eBPF programs execute in response to specific kernel events and undergo rigorous safety verification before deployment, preventing potential system crashes or security vulnerabilities. While eBPF excels in kernel-level analysis, it may not provide optimal coverage for applications requiring detailed user-space event analysis.

FTrace (Function Tracer) operates as an integrated component of the Linux kernel, focusing exclusively on the needs of the kernel developer through the comprehensive tracing capabilities at the kernel level [11]. FTrace targets scheduler behavior, driver operations, memory management, and block layer activities but cannot capture user-space application events. While FTrace can generate substantial trace data volumes, it offers precise control over tracing configuration to minimize performance impact.

2.2.2 Profiling

Profiling represents a performance analysis technique that systematically tracks statistical performance metrics during program execution, providing aggregate insights into computational resource utilization patterns [1]. Profilers identify computationally demanding code sections by extracting statistical metrics including function call frequencies, CPU time distribution, cache hit rates, and memory access patterns. These tools focus on identifying general performance trends and bottlenecks rather than providing detailed execution flow analysis.

The fundamental distinction between profiling and tracing lies in their analytical granularity and data collection approaches. Profiling provides high-level statistical summaries suitable for identifying performance hotspots and resource utilization patterns, while tracing captures detailed event sequences that enable precise execution flow analysis [34]. Profilers typically impose lower runtime overhead compared to comprehensive tracing approaches, making them suitable for continuous production monitoring scenarios.

Statistical profiling techniques sample program behavior at regular intervals, building performance profiles without requiring comprehensive event collection. Advanced profiling systems have achieved remarkable precision, with production deployments capable of detecting performance regressions as small as 0.005% across hundreds of thousands of monitored time series [11]. However, sampling approaches may miss intermittent performance issues or subtle bottlenecks that occur between sampling intervals.

Modern profiling frameworks address multiple resource dimensions simultaneously, including CPU, memory, and GPU profiling with typical overhead of 10-20% while providing superior accuracy compared to traditional profilers [35]. Memory profiling frameworks have evolved to provide fast and extensible architectures that separate frontend instrumentation from backend processing, achieving significant speedup improvements while reducing implementation complexity [36].

2.2.3 Debugging

Debugging encompasses the systematic identification and resolution of software defects through controlled program execution analysis and state examination. A debugger provides programmers with capabilities to monitor program execution, examine memory and register contents, and modify program state to test different execution scenarios [37]. Debugging focuses specifically on identifying the root causes of incorrect program behavior, distinguishing it from performance analysis or general system monitoring.

The debugging process operates through controlled execution environments that enable developers to halt program execution at specific points, examine variable states, and trace execution paths that lead to erroneous behavior [16]. Modern debugging approaches incorporate both interactive debugging sessions and automated defect detection techniques, expanding beyond traditional breakpoint-based analysis to include comprehensive execution monitoring and anomaly detection.

Contemporary debugging research has evolved toward AI-enhanced systematic approaches, particularly for complex domains such as deep learning systems. Fault localization techniques now employ knowledge graph-based approaches combining static and dynamic information to address debugging challenges in AI/ML systems [38]. Large language models are increasingly integrated into debugging workflows, though research reveals both promise and limitations, including input order bias in LLM performance for fault localization tasks [39].

Testing and debugging represent complementary but distinct software quality assurance processes. Testing focuses on identifying the existence of defects through systematic input vari-

ation and output verification, while debugging concentrates on locating and correcting the specific code sections responsible for identified defects [17]. Effective software development processes integrate both approaches to ensure comprehensive quality assurance coverage.

2.2.4 System Metrics

System metrics represent quantitative measurements of computational resource utilization and system behavior that enable performance monitoring and capacity planning. These metrics span multiple system components including CPU utilization, memory consumption, storage access patterns, and network communication characteristics [40]. System metrics provide aggregate views of system performance suitable for operational monitoring and trend analysis.

Recent research has established comprehensive taxonomies for performance indicators, providing domain-independent frameworks for performance measurement across diverse computing environments [41]. These systematic approaches to metrics classification enable standardized measurement practices while accommodating the unique characteristics of different application domains.

CPU utilization metrics distinguish between idle and scheduled processor states, where idle periods indicate available computational capacity and scheduled periods reflect active processing workloads [14]. CPU time calculations aggregate intervals during which processors execute application or system code, providing insights into computational demand patterns and resource allocation effectiveness.

Memory allocation metrics track dynamic memory management through page allocation and deallocation events, enabling analysis of memory utilization patterns and identification of potential memory leaks or excessive consumption [42]. These metrics prove essential for understanding application memory requirements and optimizing memory allocation strategies in resource-constrained environments.

File operation metrics quantify storage system interactions through system calls including file open, read, write, and close operations [15]. Network operation metrics capture socket-based communication patterns through connection establishment, data transmission, and message reception events. These I/O metrics enable identification of storage and network bottlenecks that may impact overall system performance.

2.3 Software Instrumentation Practice

Software instrumentation practice represents a critical aspect of software development that encompasses the systematic addition of monitoring capabilities to applications. This practice has evolved significantly in recent years, transitioning from manual, ad-hoc approaches toward intelligent, automated frameworks that can operate at unprecedented scale while maintaining minimal overhead.

Application instrumentation consists of two main phases: instrumentation and management, with our focus primarily on the instrumentation phase as it directly relates to the fundamental challenge of where, what, and how to instrument software systems. According to established frameworks, application instrumentation encompasses three essential steps: logging approach selection, logging utility integration, and instrumentation code composition.

The logging approach addresses cross-cutting concerns since instrumentation codes are scattered across applications and mixed with program logic. Logging utility integration involves selecting appropriate tools and configuring them for specific system requirements. Instrumentation code composition represents the most critical step, requiring developers to address three fundamental questions: where-to-log (finding appropriate positions), what-to-log (determining information content), and how-to-log (developing and maintaining instrumentation code).

2.3.1 Instrumentation Usability

The usability of instrumentation systems has become increasingly important as software systems grow in complexity and scale. Traditional approaches relied heavily on developer intuition for instrumentation placement decisions, often resulting in either excessive overhead or insufficient monitoring coverage [43]. Recent research has addressed these challenges through sophisticated cost-aware frameworks and automated assistance tools that significantly reduce the burden on developers while improving instrumentation effectiveness [23, 44].

Cost-Aware Framework Evolution

The foundation of modern cost-aware instrumentation was established by Ding et al. with Log2, a framework that operates under predefined bandwidth budgets representing the maximum volume of logs permitted within specific time intervals [22]. Log2's two-phase filtering mechanism—local filtering for discarding trivial requests and global filtering for managing top-ranked requests—reduces overhead compared to single-phase centralized approaches. The system employs dynamic utility scoring based on histogram analysis of monitored code re-

gions, with global thresholds adjusted dynamically to accommodate varying environmental conditions.

Building on this foundation, Zhao et al. developed Log20, which maximizes logging informativeness while minimizing overhead through Shannon Entropy calculations [23]. Their approach evaluates logging statements' ability to disambiguate program execution paths, using execution time as the cost function. The evolution from their initial brute-force search to dynamic programming methodology demonstrates the progression toward more scalable solutions, though they acknowledge that this approach does not guarantee optimal placement configurations in all cases.

Log4Perf extends cost-aware instrumentation specifically to web-based systems for performance analysis [44]. The framework's three-step process—identifying performance-influencing web requests, methods, and basic code blocks—employs statistical significance testing with linear regression models. This systematic approach ensures that only statistically significant metrics (P-value < 0.05) influence instrumentation decisions, providing a principled foundation for performance-oriented instrumentation.

2.3.2 Instrumentation Diagnosability

Instrumentation diagnosability focuses on how effectively instrumentation supports failure diagnosis and performance analysis. This area has experienced significant advancement through the integration of AI-enhanced techniques and systematic approaches to complex distributed system analysis.

Failure Diagnosis Enhancement

Traditional failure diagnosis approaches employed both static and dynamic analysis methods. Static analysis-based methods analyze applications without execution, using representations such as abstract syntax trees or call graphs to identify potential instrumentation locations. Yuan et al. investigated 250 bug reports to extract exception patterns requiring additional logging, while SmartLog leveraged data mining algorithms for automatic logging code instrumentation [37].

Dynamic analysis approaches follow systematic three-step processes: running applications under various settings (often with fault injection), analyzing generated data to assess current instrumentation effectiveness, and updating instrumentation decisions based on analysis results. Cinque et al. demonstrated this approach by injecting faults into open-source applications and summarizing frequently executed functions from failure scenarios [45].

Modern AI-enhanced approaches have revolutionized failure diagnosis capabilities. LogUpdater provides the first automated framework for detecting and repairing logging statement defects, identifying four critical defect types: statement-code inconsistency, static-dynamic inconsistency, temporal relation inconsistency, and readability issues [46]. The two-stage framework combining similarity-based classifiers with LLM-based recommendation systems achieves an F1 score of 0.625 in defect detection with a 61.49% success rate in automated recommendations.

The Defects4Log benchmark establishes comprehensive taxonomy for logging code defects with seven patterns and 14 scenarios, incorporating 164 developer-verified real-world logging defects [47]. This systematic approach enables up to 10.9% improvement in defect detection when domain knowledge is properly incorporated into LLM-based analysis systems.

Performance Analysis in Distributed Systems

Performance analysis has evolved to address the complexities of modern distributed architectures. Traditional logging approaches often prove insufficient for systems containing hundreds of microservices due to lack of contextual information needed for event ordering reconstruction. This challenge has driven the development of causality tracking-based solutions.

Schema-based techniques correlate relevant logs using predefined rules, requiring developers to generate event rule schemes for reconstructing complete requests through happened-before relationships [48]. Propagation-based techniques maintain causality between components by propagating metadata, with complete trace data linked through global TraceIDs and ParentIDs to preserve causality relationships across distributed components.

Recent advances have introduced network-centric approaches that eliminate traditional application instrumentation requirements. DeepFlow implements distributed tracing through network-level information, leveraging service dependency mapping and performance analysis without requiring code modifications [49]. TraceWeaver enables distributed request tracing without application modification by reconstructing traces using production timestamps and test environment call graphs, addressing the primary barrier to distributed tracing adoption [50].

Production studies reveal significant performance implications of distributed tracing. Research demonstrates that OpenTelemetry and Elastic APM frameworks can cause throughput reductions of 19-80% and latency increases up to 175%, with trace data serialization identified as the primary overhead source [31]. These findings underscore the importance of efficient instrumentation design for production deployments.

2.3.3 Instrumentation Automation

The emergence of automated instrumentation represents a paradigm shift from manual approaches toward intelligent systems capable of making sophisticated instrumentation decisions with minimal human intervention. This evolution has been driven by advances in machine learning, compiler integration, and the growing complexity of modern software architectures.

Recent advances have introduced AI-powered automated instrumentation that dramatically reduces developer burden. UniLog, presented at ICSE 2024, represents the state-of-the-art in automated logging statement generation, achieving 76.9% accuracy in logging position prediction using large language models with in-context learning [51]. This approach requires only five demonstration examples without model fine-tuning, reducing computational requirements by 96% compared to traditional fine-tuning approaches.

The FastLog framework addresses real-time integration challenges through a two-stage approach combining token classification for position prediction with sequence-to-sequence models for content generation [52]. This framework enables practical deployment in development environments, marking a crucial transition from research prototypes to production-ready tools.

Go Static advances beyond single-method contexts by incorporating inter-method static analysis, achieving significant improvements: 8.7% in position accuracy, 32.1% in level accuracy, and 138.4% in text BLEU-4 scores [53]. This work demonstrates that expanding contextual understanding dramatically enhances automated instrumentation decision quality.

Machine Learning-Driven Automation

Large language models have transformed automated instrumentation generation capabilities. The LogBench dataset provides the first standardized benchmark with 3,870 methods and 6,849 logging statements, enabling systematic evaluation of 13 top-performing LLMs ranging from 60M to 405B parameters [54]. Despite impressive progress, the maximum BLEU score of 0.249 indicates substantial opportunities for improvement in generating human-quality logging statements.

LEONID advances beyond foundational systems by distinguishing methods requiring logs from those that don't, supporting multiple log statement injection within single methods with 17% accuracy for test methods [55]. The combination of deep learning with Information Retrieval techniques reduces false positives while improving developer decision-making processes.

Advanced frameworks now incorporate sophisticated context understanding. Modern systems dynamically adjust instrumentation density based on system load, implement multi-objective optimization considering information value versus performance cost, and provide intelligent sampling strategies for large-scale systems.

Compiler-Integrated Approaches

Integration of instrumentation capabilities directly into compiler infrastructure has yielded remarkable advances in both performance and functionality. The CSI Framework provides systematic budget-aware instrumentation through strategic compiler placement, enabling high-performance profiling with controlled overhead budgets [56].

Wastrumentation represents a breakthrough for WebAssembly environments, introducing the first dynamic analysis platform supporting intercession—the ability for analysis tools to modify target program behavior [57]. This language-agnostic framework enables previously impossible analyses including memoization, safe heap access, and NaN non-determinism removal while maintaining portability across WebAssembly VMs.

Aclang introduces the first fully-featured aspect-oriented programming language for C, providing comprehensive AOP support through LLVM-based compiler infrastructure [58]. This work addresses critical gaps in cross-cutting instrumentation approaches for systems programming, offering superior effectiveness compared to previous solutions.

Cloud-Native and Modern Architecture Support

The evolution toward cloud-native, microservices, and serverless architectures has driven innovative instrumentation approaches specifically designed for modern deployment models. ServerlessLLM addresses specific challenges in serverless architectures through multi-tier checkpoint loading subsystems and locality-friendly GPU serverless architectures [59].

Advanced instrumentation systems increasingly incorporate dynamic adaptation capabilities that adjust monitoring behavior based on runtime conditions and system context. Modern frameworks demonstrate intelligent sampling strategies, runtime overhead adjustment, and context-aware instrumentation optimization that adapts to varying system conditions [60,61].

Cross-platform standardization has emerged through unified frameworks such as the 3W1H categorization framework (Why, Where, What, How well to log), providing foundational taxonomy for understanding instrumentation practices across different platforms and languages [62]. This standardization enables broader adoption of advanced instrumentation techniques across diverse technology stacks and development environments.

The convergence of AI-powered automation, compiler integration, and production validation has created unprecedented capabilities in automated instrumentation generation, defect detection, and adaptive monitoring. These advances represent a fundamental shift from traditional manual approaches toward intelligent systems that can operate effectively at hyperscale while maintaining minimal overhead and providing comprehensive system visibility.

2.4 Instrumentation Costs

The systematic study of instrumentation costs has emerged as a critical research area within software performance analysis, driven by the recognition that comprehensive monitoring requires careful consideration of the multifaceted overhead introduced by tracing and logging mechanisms. Modern software systems demand sophisticated observability capabilities, yet the associated costs can significantly impact system performance, resource utilization, and operational efficiency. Research from top-tier venues spanning 2010-2024 has established both theoretical foundations for understanding instrumentation overhead and practical optimization strategies that enable production-viable monitoring with acceptable cost profiles [28, 29, 32, 63, 64].

The instrumentation cost landscape encompasses twelve distinct categories that collectively represent the full spectrum of overhead introduced by monitoring systems. These costs range from immediate runtime performance impacts to long-term organizational burdens, with modern optimization techniques achieving dramatic reductions across multiple dimensions simultaneously. Understanding both the theoretical foundations and practical implications of these costs enables informed decision-making about when, where, and how to implement comprehensive monitoring strategies.

2.4.1 Execution Time Overhead

Execution time overhead represents the most fundamental cost function in instrumentation systems, formally defined as the additional runtime cost incurred by inserting instrumentation code into application execution paths. This overhead results from two primary factors: the execution time required to make measurements relative to event size, and the frequency of event occurrence [65]. The overhead manifests through direct execution of instrumentation instructions, context switching for trace data collection, cache disruption effects from additional code paths, and synchronization costs for thread-safe data collection.

Research has demonstrated dramatic variations in execution overhead based on instrumentation technique sophistication. HPCToolkit achieves 1-5% overhead through statistical sam-

pling approaches, while comprehensive memory tracing systems like memTrace exhibit 1.3x-3.1x overhead with geometric mean 1.97x for SPEC CPU2006 benchmarks [33]. The 2024 ISSTA study found runtime verification overhead ratios ranging from 0.3x to 5,002.9x, with some projects experiencing over 28.7 hours of absolute overhead—far exceeding the 12.48-second acceptable threshold. Additionally, 99.87% of the monitors were "wasted," highlighting the critical need for selective instrumentation strategies [4].

Zero-overhead monitoring has emerged as achievable through hardware acceleration techniques. RDMA-based approaches can achieve literally zero CPU occupation through one-sided RDMA operations [66], while eBPF instrumentation provides kernel-level efficiency with minimal userspace overhead. The Argus system demonstrates that <5% CPU overhead for system-wide tracing is achievable through careful design [67], establishing practical benchmarks for production deployment. Modern optimization techniques leverage post-domination sets to reduce counter density and minimize execution overhead while maintaining profile accuracy.

2.4.2 Concurrent Uniformity

Concurrent uniformity represents how instrumentation affects parallel system behavior and thread synchronization, though this exact terminology lacks widespread adoption in the academic literature. This cost function addresses timing perturbations affecting concurrent execution patterns, including context switching overhead, lock contention from additional synchronization points, cache disruption effects, and timing perturbation altering thread interaction patterns [68]. Measurement approaches focus on runtime verification techniques and statistical analysis comparing instrumented versus non-instrumented concurrent executions.

Related research in IEEE TPDS, OSDI, and PPoPP venues demonstrates that instrumentation introduces substantial timing perturbations affecting concurrent execution patterns. The HMTRace framework achieves 4.01% mean execution time overhead with zero false positives in race detection, using ARM v8.5-A Memory Tagging Extensions [69]. FastTrack algorithm optimizations enable O(1) space per variable in common cases, making concurrent system monitoring practically feasible.

Research reveals that concurrent systems are particularly sensitive to instrumentation overhead due to the complex interactions between threads and synchronization primitives. Demand-driven software race detection using hardware performance counters demonstrates that carefully designed instrumentation can maintain concurrent system behavior while providing comprehensive monitoring [70]. However, naive instrumentation approaches can severely disrupt concurrent execution patterns, leading to artificial race conditions or masking of real

concurrency bugs.

2.4.3 Memory Volume Overhead

Memory volume overhead encompasses buffer allocation, metadata storage, and runtime data structures required for instrumentation functionality. This cost function includes thread-local software counters for basic block tracking, ring buffers for trace collection, shadow memory for program state tracking, and metadata structures for correlation and analysis [71]. The overhead scales with both the granularity of monitoring and the duration of data retention before processing or export.

Research reveals memory overhead often exceeds CPU overhead for comprehensive monitoring, making memory-aware instrumentation design essential for production deployment [72]. The work of Kumar et al. [73] on instrumentation optimization achieved 2-3.3x performance improvements for architecture modeling through techniques like dynamic probe coalescing and partial context switching. Their approach demonstrates how optimized instrumentation can significantly reduce monitoring overhead compared to unoptimized implementations.

Buffer management critically impacts memory costs, with credit-based flow control systems managing memory efficiently while naive approaches suffer from memory leaks and unbounded growth. Static analysis techniques demonstrate remarkable improvements, with the Spindle system achieving >200x improvement over naive approaches by identifying predictable memory access patterns [74]. These findings emphasize that memory overhead requires systematic architectural consideration rather than ad-hoc mitigation strategies.

2.4.4 Disk Volume Overhead

Disk volume overhead represents storage space requirements for persistent trace data and profile information, including trace files, profile databases, log archives, and compressed data structures.

Storage costs represent an often-overlooked expense category with enormous variation based on compression and management strategies. Pattern-aware storage systems demonstrate the most promising approach for storage cost control, with LogGrep achieving order-of-magnitude cost reduction through pattern exploitation compared to naive logging approaches [75]. Industrial deployments reveal storage costs as budget-limiting factors, with Alibaba Cloud's production experience showing intelligent compression and query optimization dramatically reducing both storage requirements and query latency.

Hardware sampling approaches can reduce trace storage by orders of magnitude compared to

full software tracing, with compression techniques further minimizing storage costs. Retroactive sampling provides another approach for storage cost management, with the Hindsight system dramatically reducing storage requirements while maintaining trace accuracy through always-on collection with delayed persistence decisions [76]. Research indicates that storage costs scale non-linearly with system complexity, making intelligent aggregation and compression essential for large-scale deployments.

2.4.5 Code Size Overhead

Code size overhead measures binary size increases from instrumentation insertion, including probe instructions, control flow modifications, metadata embedding, and additional library dependencies. The overhead manifests through direct instruction insertion, additional basic blocks for instrumentation logic, increased symbol table sizes, and expanded debugging information [77]. Modern optimization techniques focus on minimizing code expansion through selective instrumentation and on-demand compilation strategies.

2.4.6 Detection Delay

Detection delay measures the time lag between event occurrence and event capture/reporting, formally defined as t_report - t_occurrence. This cost function encompasses instrumentation latency from probe execution time, data collection overhead from buffer management, network transmission delays in distributed systems, processing latency for data transformation, and buffering effects from asynchronous collection [78]. Theoretical foundations draw from queueing theory for buffering delays and network calculus for worst-case bounds.

Recent research has minimized detection delay through asynchronous processing architectures. Real-time systems now achieve microsecond-level event capture, while production monitoring systems maintain sub-second response times for performance anomalies [79]. The separation of event capture from processing, combined with lock-free data structures, has reduced detection delays by 60-80% compared to synchronous approaches.

Advanced systems achieve 590 nanosecond resolution through GPS-based synchronization, with statistical models helping predict detection delay distributions under varying load conditions [80]. Research shows that detection delay can be optimized through intelligent buffering strategies that balance latency requirements against throughput optimization, enabling real-time monitoring with acceptable responsiveness characteristics.

2.4.7 Bandwidth Overhead

The bandwidth overhead quantifies the additional network bandwidth consumed by monitoring data transmission in distributed systems, calculated as (Monitoring Traffic / Total Traffic) × 100%. This cost function includes serialization and transmission of trace data, synchronization of metadata between distributed components, control message overhead for coordination, and compression efficiency for bandwidth utilization [81]. Network overhead varies enormously based on instrumentation granularity and export frequency.

BufScope achieves <0.07% network bandwidth overhead through request-level semantic injection with SmartNIC offloading [82], demonstrating that network costs are controllable through architectural innovation. The DeltaINT framework achieves up to 93% bandwidth reduction through delta-based approaches, while zero-code tracing eliminates application instrumentation overhead through network-layer monitoring.

HeteroSketch achieves 20-60% reduction in resource overheads through programmable switch utilization [83], illustrating how network infrastructure can be leveraged to reduce instrumentation costs. However, comprehensive distributed tracing can generate substantial bandwidth requirements, with systems producing multiple megabytes per second of trace data under normal operation, highlighting the importance of intelligent data reduction strategies and hierarchical aggregation approaches.

2.4.8 Opportunity Cost

Opportunity cost represents foregone benefits when resources are diverted from feature development to instrumentation implementation and maintenance. This cost function encompasses time allocation trade-offs between monitoring and feature development, resource constraints limiting development capacity, feature prioritization conflicts, and strategic development choices affecting long-term system evolution [84]. Industry studies suggest opportunity costs can reach $10,000+ per specialized component when considering long-term maintenance overhead.

Developer productivity studies suggest significant organizational productivity impacts from instrumentation complexity, with resource allocation decisions for monitoring infrastructure competing with application development resources [85]. Research emphasizes that developers can be "five times faster for particular problems" with optimal technology choices, highlighting substantial opportunity costs of suboptimal instrumentation decisions.

Green computing research demonstrates that collecting many parameters and using accurate forecasting approaches can cause higher energy consumption than the deviation they

prevent [64], illustrating classic opportunity cost scenarios. Opportunity cost quantification remains severely understudied despite likely representing the largest total cost impact across software development organizations, requiring systematic frameworks for measuring development productivity impacts and resource allocation efficiency.

2.4.9 Maintenance Cost

Maintenance cost encompasses ongoing resources for sustaining instrumentation functionality throughout system lifecycles, including debugging instrumentation failures, updating monitoring code with application changes, managing data retention policies, and maintaining analysis infrastructure [86].

Research reveals significant human costs in instrumentation deployment and maintenance, with surveys of 66 developers plus 223 logging-related issue reports identifying performance overhead and log decision complexity as major developer concerns [1]. The DPLOG approach demonstrates >500% overhead reduction while maintaining monitoring effectiveness, but requires developer training and tool adoption, illustrating the ongoing maintenance burden.

Kieker framework research extensively documents maintenance overhead, showing that original developers may need reassignment from future projects to debug instrumentation components, resulting in unplanned delays and cascading project impacts [87]. Research shows 53% of studies focus on preprocessing costs, highlighting the significant analysis burden associated with instrumentation data management. Limited quantitative models exist for development productivity impact, despite qualitative evidence of substantial human costs in instrumentation deployment and maintenance activities.

2.4.10 Analysis Cost

Analysis cost represents computational resources and infrastructure expenses for processing collected data into actionable insights, including data processing pipelines, storage infrastructure for analytical workloads, query execution overhead, and visualization system requirements [88].

2.4.11 Energy Cost

Energy cost measures additional power consumption during logging, tracing, and monitoring operations, with formal definitions including Power Profiling (P_monitoring - P_baseline) and Time-Energy models ($\int P\_additional(t)dt$). Contributing factors span CPU cycles for

data collection, memory buffering overhead, I/O operations for trace writing, and network transmission costs [89].

Fine-grained energy measurement systems achieve 97% accuracy with <3% compute time overhead [90], enabling precise energy accounting for instrumentation decisions. Research reveals that battery life can be doubled through adaptive beaconing and intelligent sampling in mobile applications, emphasizing the importance of energy-aware instrumentation strategies for resource-constrained environments. Symmetric encryption typically imposes 5-15% CPU overhead while asymmetric operations cost 100-1000x more, highlighting the energy implications of security-enhanced monitoring.

2.4.12 Security Cost

Security cost encompasses computational, storage, and transmission overhead for implementing security measures and privacy protection in instrumentation systems. Formal definitions include cryptographic overhead and privacy protection cost for anonymization and access control [91]. Homomorphic encryption imposes 10,000-100,000x computational overhead while differential privacy adds minimal computation but affects data utility.

Security instrumentation presents unique cost challenges, balancing protection requirements with performance constraints. Origin-sensitive Control Flow Integrity achieves 7.6% average overhead while providing 98% reduction in equivalence class sizes [92], demonstrating that security monitoring can achieve reasonable performance costs with proper implementation.

Binary instrumentation for security shows 2-27% overhead depending on security guarantees provided, while comprehensive I/O monitoring for ransomware detection can incur up to 350% execution time increase [93]. However, multi-staged approaches demonstrate order-of-magnitude overhead reduction, and hardware-assisted security monitoring achieves 0.9% average performance overhead with dedicated hardware acceleration. The 2024 Meta Engineering implementation demonstrates practical large-scale cryptographic monitoring, while recent research provides comprehensive frameworks for encrypted audit trails with searchable encryption capabilities.

2.5 Conclusion

Despite significant advances in performance analysis and software instrumentation, several challenges continue to limit the practical deployment of monitoring in production systems. The field lacks systematic frameworks for understanding and managing the multifaceted costs associated with performance monitoring, and existing research mostly addresses individual

cost dimensions in isolation rather than considering their complex interdependencies and cumulative impact on system operation. Current approaches to instrumentation placement and configuration rely heavily on developer intuition and ad-hoc decision-making processes, lacking principled optimization methodologies that can automatically balance monitoring effectiveness against resource constraints across diverse operational environments. The scalability issue in modern software systems, where comprehensive monitoring can generate large amount of data annually, is not adequately addressed, with limited research on intelligent data reduction techniques that preserve analytical utility while reducing storage, transmission, and processing overhead. Furthermore, there exists a significant gap between theoretical cost-aware frameworks proposed in academic literature and their practical applicability in real-world production environments, where multiple competing constraints and dynamic operational conditions challenge the assumptions underlying many proposed solutions. Finally, the field suffers from insufficient empirical validation of whether current monitoring practices actually improve system reliability and debugging effectiveness, with most research focusing on technical feasibility rather than demonstrating measurable improvements in operational outcomes.

This thesis addresses several of these critical gaps through an investigation of cost-aware software instrumentation that bridges theoretical foundations with practical deployment requirements. The multi-dimensional cost taxonomy and optimization framework developed in this work provides the first comprehensive approach to understanding and managing the full spectrum of instrumentation costs, moving beyond fragmented single-dimension analyses toward cost-aware decision-making. The machine learning-driven approaches to trace data volume reduction directly tackle the scalability challenge by demonstrating that intelligent feature selection can maintain analytical accuracy while dramatically reducing data collection and processing overhead. The large-scale empirical analysis of logging practices provides essential evidence about the actual relationship between instrumentation and debugging across different software domains, establishing empirical foundations that have been largely absent from existing literature. Collectively, these contributions advance the field toward systematic, evidence-based approaches to cost-aware performance analysis, providing both theoretical frameworks and practical methodologies that enable effective monitoring at scale while respecting the diverse operational constraints encountered in modern software systems.

CHAPTER 3 OVERVIEW

This thesis is organized around three articles that are presented in Chapters 4, 5, and 6. Each of them addresses a research objective that is introduced in Chapter 1.

The main goal of this thesis is to improve the landscape of identifying and minimizing the costs associated with performance analysis in software systems. As software architectures have evolved from monolithic systems to distributed microservices serving billions of users, the challenge of maintaining comprehensive performance visibility while operating under strict cost constraints has intensified. Most software instrumentation research has focused on addressing fundamental questions of where to instrument, what information to capture, or how to implement monitoring mechanisms. However, only a limited number of studies have addressed the critical challenge of balancing instrumentation to achieve high-quality diagnostic data while minimizing associated costs. This thesis addresses the fundamental challenge of systematic instrumentation placement in software performance monitoring through a comprehensive cost-aware framework that bridges theoretical foundations with practical deployment requirements.

The first paper, presented in Chapter 4, "Balancing Costs and Insights: A Framework for Cost-Aware Software Instrumentation," establishes the theoretical and practical foundations for systematic instrumentation decision-making by addressing the fundamental question of where to instrument when monitoring software systems. Through an extensive literature survey examining papers from 2000-2024, this work identifies and categorizes twelve distinct cost dimensions associated with software instrumentation, ranging from immediate execution overhead to deferred maintenance burden. This multi-dimensional cost taxonomy represents the first systematic categorization that considers factors beyond execution time alone, examining costs across intrinsic versus extrinsic, operational versus non-operational, and immediate versus deferred dimensions. Building upon this cost understanding, the research develops an optimization-based framework that automatically identifies near-optimal instrumentation configurations for specified monitoring objectives while respecting cost constraints. The framework formulates instrumentation placement as a knapsack optimization problem, enabling practitioners to precisely determine instrumentation requirements while automatically selecting optimal code sections based on multiple cost constraints simultaneously. Evaluation across three diverse applications—a SPEC CPU benchmark, the Redis

database, and the Apache HTTPD web server—demonstrates that the framework effectively identifies high-variability functions that warrant monitoring attention, with selected functions consistently ranking within the top 12-27% based on actual execution time variability measurements.

The second paper, presented in Chapter 5, "Automatic Reduction of Execution Trace Data Volume Using Gradient Boosting in Large-Scale Microservice Systems," addresses a critical component of the instrumentation cost spectrum: the analysis cost associated with processing massive trace data volumes. This work tackles the scalability challenges of comprehensive system monitoring when trace data volumes become prohibitively large for analysis and storage. Using a comprehensive production dataset from Alibaba that spans 13 days in 40,000+ bare metal nodes and 470,000+ containers with 28,000+ microservices, the research develops a two-phase machine learning approach that leverages gradient enhancement algorithms to identify the most critical features for accurate CPU and memory utilization prediction. Through systematic application of gradient boost algorithms, the study demonstrates that the precision of performance modeling can be maintained or even improved while reducing the volume of trace data by more than 69%. The evaluation achieves RMSE scores of 0.02 for CPU utilization and 0.13 for memory utilization compared to 0.08 and 0.14 achieved with the full dataset, representing both significant data reduction and improved prediction accuracy. The analysis reveals that the most influential features for resource utilization modeling are closely tied to inter-service communication, memory access patterns, and database interactions, validating that targeted data collection focusing on critical architectural aspects can substantially decrease monitoring overhead without sacrificing analytical capability.

The third paper, presented in Chapter 6, "Logging Practices in Software Bug Resolution: An Empirical Study," establishes essential empirical foundations for understanding the actual effectiveness of current logging practices in software debugging contexts. This comprehensive mixed-methods investigation analyzes more than 572,000 functions in ten major open-source Java projects, combined with survey responses from 58 professional developers. The study provides the first large-scale empirical evidence of how logging patterns relate to bug resolution activities, revealing significant domain-dependent variations and challenging common assumptions about logging effectiveness. The analysis uncovers that the associations of logging practice with bug resolution vary substantially between software domains, with infrastructure projects showing strong statistical associations (odds ratios 1.21-1.48) while framework projects demonstrate minimal associations (odds ratios near 1.0). Furthermore, research identifies a significant gap between developer perceptions and actual practices: While 74%

of industry developers claim to follow balanced logging approaches, code analysis reveals predominantly reactive patterns where developers add logging 1.2-3.8 times more frequently during bug fixes than regular development. These findings establish baseline patterns for understanding when and how logging actually contributes to the effectiveness of debugging, providing essential empirical grounding for cost-aware instrumentation decisions.

In combination, these articles try to improve the field of cost-aware performance analysis in software systems by providing both theoretical frameworks and practical methodologies that enable effective monitoring at scale while respecting the diverse operational constraints encountered in modern software systems. Collectively, these contributions advance the field toward systematic, evidence-based approaches to cost-aware performance analysis, addressing the critical gaps that have hindered the adoption of comprehensive monitoring in production systems.

CHAPTER 4 ARTICLE 1: BALANCING COSTS AND INSIGHTS: A FRAMEWORK FOR COST-AWARE SOFTWARE INSTRUMENTATION

Authors: Amir Haghshenas, Naser Ezzati-Jivan, Michel Dagenais

Venue: Journal of Software: Evolution and Process

Submission Date: 03-11-2025

Abstract: Application instrumentation is essential for software performance monitoring, but the associated costs—including execution time overhead, memory consumption, and storage requirements—can significantly impact system performance. Current instrumentation practices rely on ad-hoc developer decisions, leading to either excessive overhead or insufficient monitoring coverage. This study presents a systematic approach to cost-aware instrumentation that addresses the fundamental question of where to instrument when monitoring software systems. Our primary contribution is a multi-dimensional categorization of instrumentation costs, spanning intrinsic versus extrinsic, operational versus non-operational, and immediate versus deferred cost types. Building on this taxonomy, we develop an optimization-based framework that automatically identifies near-optimal instrumentation configurations for specified monitoring objectives while respecting cost constraints. We implement a prototype that combines static code analysis and dynamic performance metrics to predict function execution time variability, then formulates instrumentation placement as an optimization problem. Evaluation on three applications—a SPEC CPU benchmark, Redis database, and Apache HTTPD web server—provides evidence that our approach effectively identifies high-variability functions: selected functions consistently ranked within the top 12-27% based on actual execution time variability measurements.

4.1 Introduction

Performance monitoring and application instrumentation are fundamental to optimizing software systems and ensuring their efficient operation [28, 33]. With increasing complexity in modern software architectures—including microservices, distributed systems, and cloud-native deployments—and growing demand for real-time monitoring, effective performance monitoring has become more crucial than ever. Modern DevOps and Site Reliability Engineering (SRE) practices have elevated monitoring from an optional development aid to a critical operational requirement, where comprehensive observability can mean the difference between proactive issue resolution and costly system failures. These developments enable

developers to identify bottlenecks, understand system dynamics, and make informed optimization decisions. However, comprehensive monitoring poses significant challenges due to the intricate nature of software systems and high expectations for real-time response.

Execution tracing and logging, which involve instrumenting software to capture critical information about behavior and performance during runtime, provide invaluable insights into system operations. However, these practices introduce costs beyond computational overhead, including increased memory usage and additional I/O operations, directly impacting application performance. This necessitates careful consideration when designing performance monitoring frameworks. The impact of these costs becomes particularly acute in production environments where even small performance degradations can cascade into significant service disruptions. For instance, high-frequency trading systems cannot tolerate additional microsecond latencies from instrumentation overhead, while mobile applications must carefully balance monitoring capabilities against battery consumption.

While existing research has provided valuable guidance on optimal instrumentation placement for specific objectives [22, 94], a critical systematic gap remains: current approaches typically focus on individual cost dimensions or single monitoring objectives without providing frameworks for understanding and managing the full spectrum of instrumentation costs [32, 63]. Most existing placement suggestion tools optimize for execution time overhead alone, neglecting memory consumption, storage requirements, or maintenance burden. Others target specific use cases such as web application performance or fault diagnosis, but fail to generalize across different monitoring objectives or application domains. This fragmented approach leaves developers without principled methods for making informed trade-offs when multiple cost constraints must be satisfied simultaneously [95].

This limitation is particularly evident in industry practice, where developers often encounter challenges when instrumenting their applications due to the lack of systematic approaches that can balance multiple cost constraints. The ad-hoc approach commonly used by developers in practice can lead to inconsistencies and inefficiencies in the instrumentation process, ultimately hindering the overall effectiveness of performance monitoring. Enabling excessive instrumentation points can result in a cluttered dataset filled with irrelevant information, making it difficult and costly to identify and resolve problems. Conversely, enabling too few instrumentation points may lead to the omission of crucial data, disrupting the ability to effectively address performance issues [43].

This challenge becomes more pronounced in two specific scenarios. Firstly, in large software

systems with a significant number of function definitions that are potential candidates for instrumentation, selecting the most suitable instrumentation points becomes a complex task. Secondly, limited resources, such as constrained memory, bandwidth, or real-time response requirements, pose additional challenges for effective instrumentation. In embedded systems or IoT devices, where resources are often scarce, careful consideration must be given to balancing the instrumentation overhead with the desired level of performance monitoring. The challenge lies in striking a balance between capturing sufficient details for effective performance monitoring and minimizing the associated costs.

This research addresses these limitations by introducing a systematic cost-aware instrumentation framework designed for software performance monitoring. Our work makes several key contributions to the field: First, we provide a multi-dimensional categorization of instrumentation costs, offering the first systematic taxonomy that considers factors beyond execution time alone and examines their interrelationships. Second, we present a mathematical formulation of the cost-aware instrumentation problem that enables practitioners to precisely determine instrumentation requirements while automatically identifying optimal code sections based on multiple cost constraints. Third, we develop a prototype implementation that demonstrates the ability of our framework to identify near-optimal instrumentation placement across different application types using statistical analysis and optimization techniques.

Through these contributions, our research offers practical solutions for researchers and practitioners in software monitoring and analysis.

The remainder of this paper is organized as follows: Section 4.2 presents research studies related to cost-aware instrumentation, software performance monitoring, and instrumentation practices. Section 4.3 explains our cost categorization and mathematical formulation of the instrumentation framework. Section 4.4 outlines the implementation steps required to apply our framework in real-world applications. Section 4.5 presents the methodology we used to develop and evaluate a prototype of our framework, along with the results of each step of our methodology. Section 4.6 discusses threats to the validity of our study. Section 4.7 explains our discussion around our findings and compares our approach with related work. Section 4.8 presents the conclusion and directions for future work.

4.2 Related Works

Within this section, we categorize the previous studies pertinent to our research into three distinct aspects. 1. Cost-Aware Instrumentation 2. Software Performance Monitoring 3. Software Instrumentation Practice.

4.2.1 Cost-Aware Instrumentation

Several research studies have focused on evaluating, representing, and managing the execution time overhead associated with system tracing. Gebai et al. [29] conducted a survey that compared multiple tracers by analyzing the execution time of a single trace point in both kernel space and user space. They provided an extensive overview of different tracers, highlighting their features and distinctions. Additionally, they developed a kernel module to repeatedly execute an empty trace point and measured hardware performance counters before and after each trace point to observe the impact of various metrics on the execution time of the trace point.

Tchamgoue et al. [63] conducted a study focused on time-aware instrumentation for hard real-time applications. Their methodology involved three key steps: first, they created a call graph from the functions of the application to understand its structure. Next, they extracted execution time information using static code analysis, gaining insights into how different functions contributed to overall execution time. Finally, they proceeded to instrument the selected subgraph of the application.

In the realm of tracing studies, Orton et al. [32] explored different costs for tracing. They specifically analyzed the impact of tracing on compute volume (added CPU cycles), memory volume (tracer memory consumption), and uniformity (concurrent behavior of multi-threaded applications). Their findings revealed that while tracing may be uniform across different threads, the results of tracing might not be uniform due to variations in thread behavior.

Arafa et al. introduced QDIME [96], aiming to guarantee Quality of Service (QoS) metrics provided by application developers. To achieve this, they adopted a similar approach to their previous tool, DIME [97], which involved periodically enabling and disabling instrumentation to meet the QoS properties of the application. Such efforts in exploring different costs and ensuring Quality of Service metrics are valuable in the context of effective and efficient tracing mechanisms for software systems.

While the existing body of work has enhanced our understanding of specific costs associated with system instrumentation, such as execution time overhead, memory consumption, a framework that encapsulates the full spectrum of instrumentation costs remains absent. Our study seeks to fill this gap by presenting a unified framework that not only considers these individual cost elements but also examines their interrelations and collective impact on system instrumentation practice.

4.2.2 Software Performance Monitoring

The domain of software performance monitoring encompasses three primary categories of techniques. The first category, system monitoring, involves observing the operational status of a running software system using performance counters obtained from the underlying infrastructure. Performance counters such as CPU usage, memory usage, and I/O traffic provide valuable insights into system performance, resource allocation, capacity planning, and crash prediction [40] [98]. However, due to limited knowledge of the internal workings of the software, leveraging this data for fine-grained improvements at the source code level poses challenges.

The second category encompasses extensive tracing techniques that record every function call during system execution. Researchers have leveraged this tracing information to enhance system quality and efficiency [99] [100]. Tools like JProfiler are commonly used to generate such tracing information. However, the significant overhead introduced by tracing tools hinders their use in large-scale production environments, limiting their adoption primarily to development environments. Nonetheless, efforts have been made to extract meaningful patterns from tracing information for performance investigations and system improvement [101] [102].

The third category revolves around logging techniques, where developers insert logging statements within the source code to capture valuable information about runtime system behavior. Logging statements typically include a log level, a static text representing the logged event, and variables providing context. These logs serve as crucial sources of information for debugging and maintaining large software systems, offering flexibility and insights based on the knowledge of developers [103] [104].

Our cost-aware framework builds upon these monitoring approaches by providing systematic guidance on where to instrument for effective performance monitoring while respecting cost constraints.

4.2.3 Software Instrumentation practice

The first step in determining the best instrumentation location (where-to-log) involves developers relying on their instincts, as shown by numerous studies [27, 105]. However, striking the right balance is challenging: excessive instrumentation can lead to performance overhead, while inadequate instrumentation results in vital information being missed, making diagnosis difficult [37].

In recent years, several studies have addressed challenges in instrumentation code composition, focusing on usability, diagnosability, and code quality. For example, Ding *et al.* introduced Log2 [22], a cost-aware framework for logging with a predefined budget, employing local and global filtering phases to reduce overhead. Similarly, Zhao *et al.* developed Log20 [23, 106], which maximizes the informativeness of logging statements while minimizing overhead by calculating Shannon Entropy [107] and considering execution time as a cost.

Another framework, Log4Perf [44], focuses on suggesting log placements in web-based systems for performance analysis. It follows a three-step process: identifying performance-influencing web requests, methods, and basic code blocks. By performing performance tests, monitoring metrics, and applying statistical analysis, Log4Perf identifies significant performance influencers and suggests logging placements accordingly.

Recent advances in automated logging have leveraged deep learning techniques to improve instrumentation placement decisions. Du et al. introduced DeepLog [108], a deep learning-based approach for log location recommendation that extracts semantic and syntactic features from abstract syntax trees to train neural network models. DeepLog demonstrates significant improvements in log placement recommendation, achieving F1 scores that are 28.17% higher than existing approaches across 23 software projects.

Instrumentation Diagnosability

Within the diagnosability category, which means how instrumentation can help diagnose performance problems, failure diagnosis poses a significant challenge. Various methods have been proposed to address this, including program-analysis techniques such as static and dynamic analysis. Static analysis involves examining the code representation of the application, like abstract syntax trees or call graphs, without executing the code. This approach can detect code deficiencies useful for identifying instrumentation locations. Yuan *et al.* conducted a study using static code analysis to enhance failure diagnosis by identifying exception patterns

that require additional logging [37]. SmartLog utilizes data mining algorithms to automatically instrument logging code snippets. It analyzes logging regions to generate log intention models, aiding in determining whether specific regions should be logged or not.

Several studies have employed dynamic analysis to suggest logging points. For instance, Cinque *et al.* [109] injected faults into open-source applications, collected log messages and memory dumps, and then added additional logging code to frequently executed functions. Crameri *et al.* [110] and Jia *et al.* [111] pursued similar approaches, respectively focusing on determining logged branches and facilitating fault detection through comparison of log outputs.

Another challenge in the field of diagnosability is performance analysis, particularly in complex software systems with multiple interacting components. Traditional logging approaches may prove insufficient in such cases due to a lack of contextual information for reconstructing event ordering. Causality tracking-based solutions offer a resolution, employing two approaches:

- *Schema-based techniques:* This technique correlates the relevant logs based on some pre-defined rules [112, 113]. In this technique, developers need to generate some event rule schemes to join the individual events in order to reconstruct the complete request. This is usually done by using the happened-before relation.
- *Propagation-based techniques:* This technique keeps track of causality between components by propagating the metadata between instrumented components and a complete trace of log data is linked by the metadata. When the metadata is propagating between the components, different information is assigned to it, such as a *global TraceID* and a *ParentID*, to keep the causality relation between each component.

Despite the valuable progress in log placement suggestion for various purposes, such as performance analysis, usability, and diagnosability, the consideration of the associated costs of instrumentation has been missing in the mentioned studies. While previous works have focused on refining the technical aspects of logging and its application to specific system challenges, our framework distinguishes itself by considering the costs that come with instrumentation. This ensures that system performance and efficiency are balanced with cost-effectiveness, filling a critical gap in the landscape of cost-aware instrumentation research.

4.3 Cost-Aware Instrumentation Framework

In this section, we present the conceptual model of our proposed cost-aware instrumentation framework. We outline its fundamental attributes, including instrumentation cost categorization and underlying optimization objectives, and demonstrate the mathematical formulation of the problem.

Before elaborating on the framework description, it is essential to establish the intention of such a framework. As noted in Section 4.1, tracing and logging represent predominant approaches for monitoring system performance. The term monitoring has been used with varying connotations in the literature. Our definition of performance monitoring aligns with that provided in [114] and diverges in approach and emphasis from [115]. Throughout this article, performance monitoring is defined as the collection of information to gain insight into application resource utilization, such as CPU and memory, to establish an overview of application runtime behavior.

The framework presented in this research systematically addresses the fundamental question of where-to-instrument when initiating performance monitoring for new applications. Rather than relying on ad-hoc developer decisions, our proposed framework employs optimization-based analysis to identify a subset of functions as primary candidates for instrumentation. This systematic approach facilitates the collection of the most relevant monitoring information while respecting predefined cost constraints across multiple dimensions. By balancing monitoring effectiveness against instrumentation overhead, this approach enables developers to obtain an understanding of their applications' performance characteristics, supporting informed decisions regarding bottleneck identification, performance optimization, and strategic planning of system enhancements.

Consider as an example, the challenge faced when instrumenting Redis [1], an open-source in-memory data structure store comprising more than 4,100 function definitions with diverse functionalities including memory access, file access, and network operations. Monitoring Redis provides valuable insights into runtime behavior and is essential for maintaining application availability. However, determining optimal instrumentation locations presents a significant challenge that illustrates the need for systematic cost-aware approaches. Instrumenting every function would introduce unacceptable costs such as substantial execution time increases and excessive log data generation, while instrumenting too few functions risks missing

[1] www.redis.io

critical performance issues. The developer must balance multiple cost dimensions—execution overhead, memory consumption, storage requirements, and network bandwidth—against the monitoring value provided by different code sections. This trade-off decision becomes particularly complex given the multi-resource nature of Redis, combining intensive network operations, memory management, and periodic I/O activities. Such complexity demonstrates why ad-hoc instrumentation decisions are insufficient and systematic cost-aware frameworks are necessary.

The following subsections detail the components of our framework: subsection A presents our categorization of instrumentation costs across multiple dimensions, subsection B discusses how instrumentation objectives influence placement decisions, and subsection C formalizes the cost-aware instrumentation problem as a mathematical optimization framework.

4.3.1 Instrumentation Cost

Software instrumentation operations, while crucial for monitoring system performance, impose significant costs that extend beyond execution time. We conducted a systematic literature review following established methodological guidelines to present an overview of instrumentation-related costs.

We employed a multi-step strategy to achieve thorough and reproducible search. We initially defined relevant keywords including 'cost,' 'cost-aware,' 'overhead,' 'logging,' 'tracing,' and 'instrumentation,' combining these terms using Boolean operators to optimize our search across multiple academic databases: IEEE Xplore, ACM Digital Library, ScienceDirect, Scopus, and Google Scholar. We covered literature published between 2000 and 2024 to ensure both foundational and contemporary perspectives are captured.

We structured our screening process in well-defined stages with precise inclusion criteria focusing on studies that: (1) addressed cost-related aspects of instrumentation, (2) discussed data collection methodologies, (3) examined data analysis techniques, or (4) explored knowledge extraction from instrumentation data. We excluded papers that presented 'cost' merely in financial terms without addressing execution tracing and logging, focused solely on implementation details without cost considerations, or provided only high-level overviews without technical depth. To enhance thoroughness, we conducted backward and forward searches, examining reference lists and tracking citations of selected papers. We evaluated each paper based on methodological rigor, empirical validation, relevance to instrumentation costs, and clarity of cost quantification.

Our review can have limitations. Our focus on academic databases may have excluded some industry reports, and the rapid evolution of instrumentation technologies means some recent innovations may not yet appear in published literature.

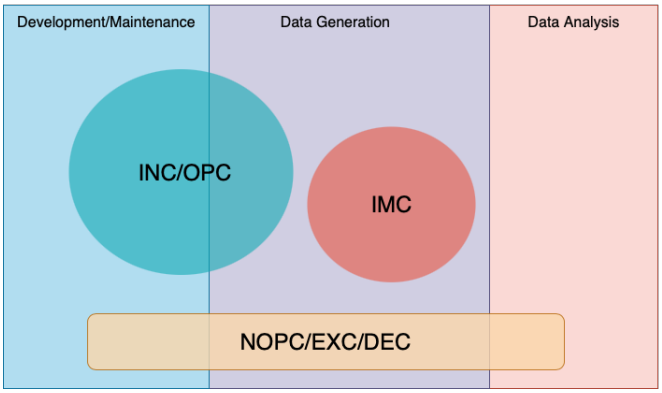

Figure 4.1 Diagram showing the relation of each cost category with instrumentation steps

The process of system logging and tracing consists of multiple stages [116]. We focus on the three key steps outlined below:

1. **Instrumentation Development and Maintenance (IDM)**, which is the process of developing the logging and tracing code and maintaining the system.
2. **Data Generation (DG)**, which is when the instrumentation code is executed and log data is generated.
3. **Data Analysis (DA)**, which is the process of analyzing the generated data in order to extract useful information and insights.

After thorough review of existing literature, we have categorized the costs related to tracing and logging into specific categories across three dimensions to provide a clear understanding of their impact on software systems. Figure 4.1 illustrates the multi-dimensional nature of these cost categories and their relationships across the instrumentation process stages.

Intrinsic and Extrinsic Costs

Intrinsic (INC) Costs: These are direct costs that arise as an immediate and necessary consequence of the instrumentation process itself. They represent the unavoidable overhead directly attributable to the addition of instrumentation code. Intrinsic costs are tightly

coupled with the act of instrumentation and would not exist without it. Examples of Intrinsic costs include execution time overhead, where additional CPU cycles are specifically required for executing the instrumentation statements [63]. Another example is code base expansion, which occurs because instrumentation inherently increases application size through added instrumentation statements [85]. Memory consumption represents another Intrinsic cost, as instrumentation requires additional memory to execute its logic and store temporary data structures during runtime [32]. Additionally, storage usage for generated logs and trace data constitutes a direct Intrinsic cost of the instrumentation process.

Extrinsic (EXC) Costs: These are indirect costs that emerge not from the instrumentation itself, but from the interaction between instrumentation and other system components or processes. They represent secondary effects that impact the broader system environment beyond the immediate instrumentation points.

Examples of Extrinsic costs include maintenance burden, where additional developer time is required to maintain instrumentation code as the system evolves [32, 85]. Privacy management represents another Extrinsic cost, as resources must be dedicated to securing and managing sensitive information captured in logs [32]. System interaction effects constitute an important category of Extrinsic costs, where performance impacts manifest in components that interact with instrumented components [22]. Furthermore, opportunity costs arise when development resources allocated to instrumentation could have been used for other system improvements [85].

Operational and Non-Operational Costs

While Intrinsic and Extrinsic costs help us understand the source of costs, a second categorization focuses on their impact on system functionality. **Operational Costs (OPC)**: These costs directly affect the runtime operation of the system [117]. They impact system performance, resource utilization, and user experience during execution. Operational costs manifest while the system is actively running and typically affect metrics that end-users and system administrators can observe.

Examples of Operational costs include increased CPU utilization, where instrumentation consumes processing resources that could otherwise be used by application logic [29, 63]. Memory overhead during execution represents another Operational cost, as instrumentation often requires additional memory for buffering and processing trace data [32]. I/O bandwidth consumption constitutes an Operational cost when instrumentation generates data that must be written to disk or transmitted over networks [63]. Additionally, energy consumption in-

creases represent an Operational cost particularly relevant in mobile or battery-powered environments [64].

Non-Operational Costs (NOPC): These costs affect aspects of the system beyond its runtime behavior. They impact system development, maintenance, and evolution without necessarily changing observable runtime characteristics. Non-Operational costs typically affect metrics that are important to developers and project managers rather than end-users. Examples of Non-Operational costs include increased code complexity, where instrumentation makes source code harder to understand and maintain [85], [28]. Development time extension is another Non-Operational cost, as adding and maintaining instrumentation requires additional engineering effort [64]. Test coverage challenges emerge as a Non-Operational cost when instrumentation code requires its own testing strategy [85]. Furthermore, cognitive load on developers increases as they must understand both application logic and instrumentation behavior, representing a significant Non-Operational cost [28].

Immediate and Deferred Costs

The third dimension of our cost categorization addresses the temporal aspect of instrumentation costs.

Immediate Costs (IMC): These costs manifest immediately upon the implementation or activation of instrumentation. They are observable as soon as instrumentation is added or enabled, and their impact is felt in the short term. Examples of Immediate costs include runtime performance degradation, which occurs as soon as instrumented code executes [29]. Memory consumption increases represent another Immediate cost that manifests as soon as instrumentation is activated [28]. Additional disk I/O operations constitute an Immediate cost when trace data is written to storage [63]. Furthermore, network bandwidth utilization increases immediately when trace data is transmitted over the network [29].

Deferred Costs (DEC): These costs accumulate over time or become apparent only after extended periods. They may not be noticeable initially but grow in significance as the system evolves or as the volume of instrumentation data increases.

Examples of Deferred costs include technical debt accumulation when instrumentation code is not maintained alongside application code [64, 85]. Log storage management becomes a Deferred cost as log data accumulates over time, requiring increasing storage capacity and

management effort [85]. Analysis complexity grows as a Deferred cost when the volume of collected data makes it increasingly difficult to extract meaningful insights [117]. Moreover, code decay represents a significant Deferred cost when instrumentation that is no longer relevant remains in the codebase, adding unnecessary complexity [118].

A crucial step towards effectively addressing these diverse costs involves understanding their distinct aspects. It is essential to acknowledge that each individual cost may belong to multiple categories and can exert various impacts on the system. Likewise, each cost category can belong to various stages in the instrumentation process. To illustrate the interconnected nature of the identified cost categories, Figure 4.1 presents a diagram that visually illustrates their overlaps. This visual representation enhances our understanding of the interrelationships among different cost categories. By observing the diagram, we can note that the Intrinsic and Operational costs extend from the development and maintenance phases to the data generation stage. Similarly, the Non-operational, Extrinsic, and Deferred costs span all three stages of instrumentation. Immediate costs, in contrast, primarily pertain to the data generation step.

Considering the various costs related to instrumentation, developers have the ability to mitigate the costs tied to tracing and logging through different strategies, and sometimes, a single approach can address several costs. For example, by limiting the number of instrumentation points, developers can reduce costs related to execution time overhead, code size, memory, and disk volume overhead [23, 94]. Automated log parsing [119] and log mining [120] algorithms can help reduce the analysis costs. Moreover, the opportunity cost can be mitigated by taking advantage of automated log statement generation techniques [88]. Optimizing the instrumentation placement can enable developers to reduce the detection delay and the code size costs when instrumenting their application [85].

It is essential to realize that some solutions may have unintended effects on other costs. Adding encryption, for instance, might affect log data generation, storage, and parsing negatively. Although different studies have proposed approaches to mitigate various costs associated with logging and tracing, the majority of these studies primarily focus on intrinsic and operational costs, as discussed in Section 4.2.1. This highlights the need for additional research on other instrumentation costs and methods to mitigate their effect. Table 4.1 offers an overview of the costs we have identified, their categories, and potential ways that, in our opinion, could help alleviate these costs.

Table 4.1 Summary of the existing costs for instrumentation.

Instrumentation Cost	Definition	Step	Category	Mitigation
Execution Time Overhead	CPU cycle to execute the instrumentation	DG	INC/OPC/IMC	Limit instrumentation
Concurrent Uniformity	Effect of instrumentation on concurrent behavior of the system	DG	EXC/NOPC/IMC	Reduce contention in threads
Memory Volume Overhead	Memory needed to collect the data	DG	INC/OPC/IMC	sample data collection/Limit instrumentation
Disk Volume Overhead	Additional disk space needed to store large data	DG	INC/OPC/IMC	use compression techniques
Code Size	Increasing code size due to added instrumentation	IDM	INC/NOPC/DEC	Optimize placement /Refactor code regularly
Detection Delay	The time lag between an event occurring and being captured	DG	EXC/NOPC/DEC	Optimize placement
Bandwidth Overhead	Bandwidth needed to transfer the data to analysis units	DG	IMC/OPC/INC	Compress/buffering data before transfer
Opportunity Cost	Resource spent on instrumenting instead of developing new features	IDM	EXC/OPC/DEC	Automate instrumentation process
Maintenance Cost	Time spent on maintaining the logging system	IDM	EXC/NOPC/DEC	Regular maintenance schedule
Analysis Cost	Time/computational resource needed for analysis of the data	DA	EXC/NOPC/DEC	Automated analysis
Energy Cost	Energy consumed by the system during logging	DG	INC/OPC/IMC	Energy-efficient hardware/software
Security Cost	Additional security measures to ensure privacy of data	DG	EXC/OPC/IMC	Minimize the collection of sensitive data

Understanding and categorizing these various costs provides valuable insights for developers and organizations when implementing a cost-aware tracing framework. Striking a balance, between the benefits of tracing and the associated costs, is vital to optimize performance monitoring and enhance overall software system efficiency.

Our categorization of potential instrumentation costs highlights the need for further investigation into the domain of cost-aware instrumentation. By delving deeper into this research area, practical insights can be gained to guide future studies and aid developers and practitioners in making well-informed decisions regarding their instrumentation practices. By considering the broader spectrum of costs, beyond just the direct tracing expenses, researchers can shed light on the complexities and nuances of cost-aware instrumentation, enabling the development of more comprehensive and effective approaches. This research serves as a catalyst for advancing the understanding and application of cost-aware instrumentation techniques, ultimately contributing to the refinement and optimization of software system performance monitoring practices.

4.3.2 Instrumentation Objective

The selection of instrumentation objectives fundamentally shapes the design and implementation of effective monitoring strategies, as different monitoring goals demand distinct approaches to code section prioritization and cost consideration. Understanding this relationship is crucial for developing systematic cost-aware frameworks that can adapt to diverse monitoring requirements.

Common instrumentation objectives span diverse system concerns. Performance monitoring seeks to identify bottlenecks, resource utilization patterns, and execution time variability to optimize system efficiency. Root cause analysis aims to pinpoint underlying causes of system failures by capturing detailed error conditions and execution flows. Security monitoring focuses on detecting unauthorized access and potential vulnerabilities through authentication events and data access patterns. Workload modeling seeks to understand system demands under various conditions to support capacity planning. Fault tolerance assessment monitors system resilience by tracking error recovery mechanisms and state transitions during failure scenarios.

Each objective significantly influences which code sections warrant instrumentation priority. For performance monitoring, code sections with high execution frequency, resource-intensive operations, or variable execution times become primary targets. Functions containing com-

plex algorithms, I/O operations, or synchronization primitives require detailed monitoring to capture performance bottlenecks. Security monitoring prioritizes functions responsible for authentication, authorization, encryption, and sensitive data handling, regardless of their performance characteristics. Root cause analysis focuses on exception handling blocks, error reporting mechanisms, and state transition points where failures commonly occur.

The relationship between instrumentation objectives and associated costs demonstrates both direct and indirect dependencies. Objectives directly influence which cost types become most critical for system operation. For performance monitoring in latency-sensitive applications, minimizing execution time overhead becomes paramount, as additional instrumentation latency would distort the measured metrics. Security monitoring in high-compliance environments may prioritize minimizing storage costs and ensuring data privacy, requiring careful consideration of captured information. Resource-constrained IoT devices implementing fault tolerance monitoring must minimize memory consumption and energy costs regardless of the specific monitoring objective.

Furthermore, deployment environment constraints often dictate cost priorities independent of monitoring objectives. High-traffic production systems may focus on reducing bandwidth overhead to prevent network contention, while embedded systems must minimize memory footprint regardless of whether they perform performance analysis or security monitoring. Cloud-based applications may prioritize storage cost reduction due to operational expense concerns, while real-time systems must minimize any overhead that could affect timing guarantees.

This relationship between objectives, code section prioritization, and cost constraints underscores the importance of a flexible framework that can adapt to various monitoring requirements. A systematic approach must enable practitioners to specify their monitoring objectives, automatically identify relevant code sections based on those objectives, and optimize instrumentation placement while respecting the cost constraints most critical to their specific deployment environment and operational requirements.

4.3.3 Framework Formulation

Building upon the cost categorization presented in section 4.3.1 and the objective-driven instrumentation approach discussed in section 4.3.2, we now formalize our framework as a systematic optimization problem. The framework integrates multiple cost dimensions with specified monitoring objectives to automatically identify near-optimal instrumentation con-

figurations that balance monitoring effectiveness against resource constraints.

Our framework operates at function-level granularity to enable precise analysis and decision-making through several key characteristics: it considers single or multiple cost dimensions based on system requirements, formalizes instrumentation as a mathematical optimization problem enabling automated decision-making, and provides a systematic method for quantifying both costs and benefits of instrumenting specific functions.

The framework transforms instrumentation placement into a structured decision-making process using a metric-based approach. We establish a method for quantifying how essential each code section is for the selected monitoring objective by assigning value scores to individual functions, while simultaneously calculating cost scores that represent the overhead introduced by instrumenting each function. This approach enables systematic comparison of different code sections and evaluation of how various cost types impact overall system performance.

The framework accommodates varying system constraints through developer-specified instrumentation budgets. These budgets represent limits on execution time, memory usage, disk storage, network bandwidth, or combinations thereof, reflecting the capacity of the system for additional costs before observing performance degradation. Our general solution adapts to both single-constraint and multi-constraint scenarios through a unified mathematical formulation.

To identify the most relevant functions while minimizing additional instrumentation costs, we formulate the problem using the well-established knapsack optimization framework [121]. The general optimization problem is presented as follows:

$$Maximize \sum_{k=1}^{N} r_k x_k$$

$$Subject\ to: \sum_{k=1}^{N} w_k x_k < C$$

Where:

- w_k = The cost of instrumenting each function for k = 1, 2, ..., N,
- r_k = The value tied to each function, for k = 1, 2, ..., N,

- C = The normalized instrumentation budget,
- x_k = The decision variables for each function.

The value score r_k is calculated as a weighted sum of metrics that indicate the importance of a function for the selected objective: This linear aggregation reflects a modeling choice: it assumes that the considered metrics contribute independently to the monitoring value of a function. We acknowledge that this assumption may not capture all possible non-linear interactions between metrics; however, this choice keeps the model transparent and easy to reproduce.

$$r_k = \sum_{i=1}^{M} \alpha_i . m_{i,k}$$

Where:

- α_i = The weight factor for metric i
- $m_{i,k}$ = The value of metric i for function k
- M = The total number of metrics considered

For single cost constraints, where only one cost type is considered (such as execution time overhead or memory consumption), we normalize the raw cost value directly against the specified budget:

$$W_k = c_k / C\_Budget$$

Where:

- c_k = The value of cost function k
- C_Budget = The total allowable budget for that specific cost type as specified by the developer.

For multiple cost constraints, common in real-world applications that face limitations across different resource dimensions, our instrumentation placement problem becomes an instance of

the 0/1 Multidimensional Knapsack Problem (MKP) [122], where each function corresponds to an item with multiple resource consumptions and a single value. MKP is a well-studied offline combinatorial optimization problem, and many exact and heuristic algorithms have been proposed, including integer linear programming [123] (ILP) formulations, branch-and-bound, and metaheuristics.

In our setting, we can therefore directly adopt the standard multi-dimensional 0/1 knapsack ILP:

$$Maximize \sum_{k=1}^{N} r_k x_k$$

Subject to:

$$\sum_{k=1}^{N} c_{j,k} x_k \leq C_j^{\text{budget}} \quad \text{for } j = 1, \ldots, P,$$

$$x_k \in \{0, 1\} \quad \text{for } k = 1, \ldots, N.$$

Here x_k indicates whether function k is instrumented, r_k is the value score and each constraint enforces one resource budget C_j^{budget} (for example, execution-time overhead, storage, or bandwidth).

The multi-dimensional formulation above is a 0/1 integer linear program and can be solved using general-purpose MIP solvers in OR-Tools. The solvers can apply branch-and-bound, cutting planes, presolve, and primal heuristics to explore the space of instrumentation configurations, providing optimal or high-quality near-optimal solutions even in the presence of multiple resource constraints.

In addition to the exact multi-constraint ILP formulation, our model also admits a normalized scalar variant that may be preferable when a simpler single-constraint knapsack is desired. In this variant, each cost $c_{j,k}$ is first normalized with respect to its budget C_Budget_j and then combined into an aggregated cost score using a weighted sum:

$$NormalizedCost_{j,k} = c_{j,k} / C\_Budget_j$$

Where:

- $c_{j,k}$ = the raw cost of type j for function k
- C_Budget_j = The budget limit specified for cost type j.

The weighting factors for combining these normalized costs are determined using the Average Normalized Cost Method, which reflects the relative constraint tightness of each resource type:

$$\beta_j = AverageNormalizedCost_j / \sum_{l=1}^{P} AverageNormalizedCost_l$$

This weighting approach ensures that more constraining resources (those with higher average normalized costs relative to their budgets) receive proportionally higher influence in the optimization decisions.

The combined cost score for each function is then computed as:

$$W_k = \sum_{j=1}^{P} \beta_j . NormalizedCost_{j,k}$$

This normalized formulation is computationally attractive and easier to integrate with existing knapsack solvers, but it should be viewed as an approximation rather than a strict enforcement of all original budgets. Because heterogeneous resource constraints are projected onto a single scalar, some individual constraints may be slightly exceeded even when the aggregated constraint is satisfied, especially when one resource is much tighter than the others. In practice, the choice between the exact multi-constraint ILP and the normalized single-constraint model is left to practitioners: engineers can select the formulation that best matches how strict their resource limits are, and may adjust the weights β_j or add explicit hard constraints whenever certain budgets must not be violated under any circumstance.

Within our framework, developers initially define the monitoring objective and specify cost constraints based on system limitations and operational requirements. The framework then automates metric extraction, cost calculation using the appropriate single or multi-constraint approach, and optimization problem solving. This structured approach ensures consistent methodology across diverse scenarios, requiring minimal manual intervention after initial

configuration and allowing developers to focus on interpreting results rather than managing the complex decision-making process.

Within our proposed framework, certain elements require manual input while others can be automated. Initially, developers are required to define the objective for instrumenting their code. Following this, the costs associated with the chosen goal are typically related to the limitations of the system.

4.4 Framework Application Guidelines

This section presents the practical implementation steps required to apply our framework in real-world software environments. Building upon the mathematical formulation presented in Section 4.3, we provide concrete guidance for practitioners seeking to deploy the framework across diverse application domains. The implementation process transforms the theoretical optimization problem into actionable steps that can be systematically followed regardless of the specific monitoring objectives or cost constraints.

The implementation process begins with defining the instrumentation objective, which determines the focus and scope of the monitoring effort. For performance monitoring, the emphasis should be on functions with high execution time variability that may indicate performance bottlenecks. Root cause analysis prioritizes code sections with high error frequency or complex exception handling mechanisms. Security monitoring necessitates instrumenting functions responsible for authentication, authorization, or sensitive data operations. The selected objective directly influences which code characteristics become most relevant for the subsequent metric extraction and value score calculation phases.

Following objective definition, implementers must identify and specify the cost functions to minimize based on system constraints and operational requirements. As detailed in our cost taxonomy (Table 4.1), these may include execution time overhead for latency-sensitive applications, memory volume overhead for resource-constrained environments, or bandwidth overhead for distributed systems. For embedded systems, energy consumption often represents the primary constraint, while systems handling sensitive information may prioritize security-related costs. The selected cost types directly influence the subsequent cost measurement and budget specification phases.

The next step involves quantifying the value of each code region relative to the selected objective through systematic metric extraction. This process implements the value score calculation (r_k) from our mathematical framework by collecting relevant static and dynamic metrics from the application code. Performance monitoring requires metrics such as execution time variability, loop counts, and call frequency, while root cause analysis benefits from metrics including exception density, code complexity, and change frequency. These metrics are combined using appropriate weight factors to produce consolidated value scores for each function, implementing the weighted sum formulation presented in Section 4.3.

Similarly, the implementation requires calculating instrumentation costs for each code region to enable the cost score computation (w_k) from our framework. This involves measuring the overhead introduced by instrumenting each function, such as additional execution time for trace point execution or memory footprint for data collection. The cost calculation must account for the specific instrumentation approach being used and the target deployment environment characteristics.

The final step involves formulating and solving the optimization problem using the calculated value and cost scores, implementing the knapsack formulation presented in Section 4.3. For single-constraint scenarios, standard dynamic programming algorithms provide optimal solutions with guaranteed optimality. For large-scale applications or multi-constraint scenarios, genetic algorithms offer efficient near-optimal solutions with acceptable computational overhead. The selection between these approaches depends on requirements for optimality versus computational efficiency, application size, and available processing resources.

Implementation of these guidelines enables effective application of our framework across diverse software domains while ensuring optimal monitoring coverage within specified system constraints. The systematic approach ensures reproducible results and provides a principled alternative to ad-hoc instrumentation placement decisions.

To address the example of Redis instrumentation challenge that we presented in Section 4.3 using our cost-aware framework, the developer would follow the workflow illustrated in Figure 4.2. First, they would define their monitoring objective, such as maximizing performance visibility, minimizing critical issue detection time, or optimizing resource utilization monitoring. Next, they would identify the relevant system limitations and select appropriate cost functions from the taxonomy of our framework, potentially including execution time

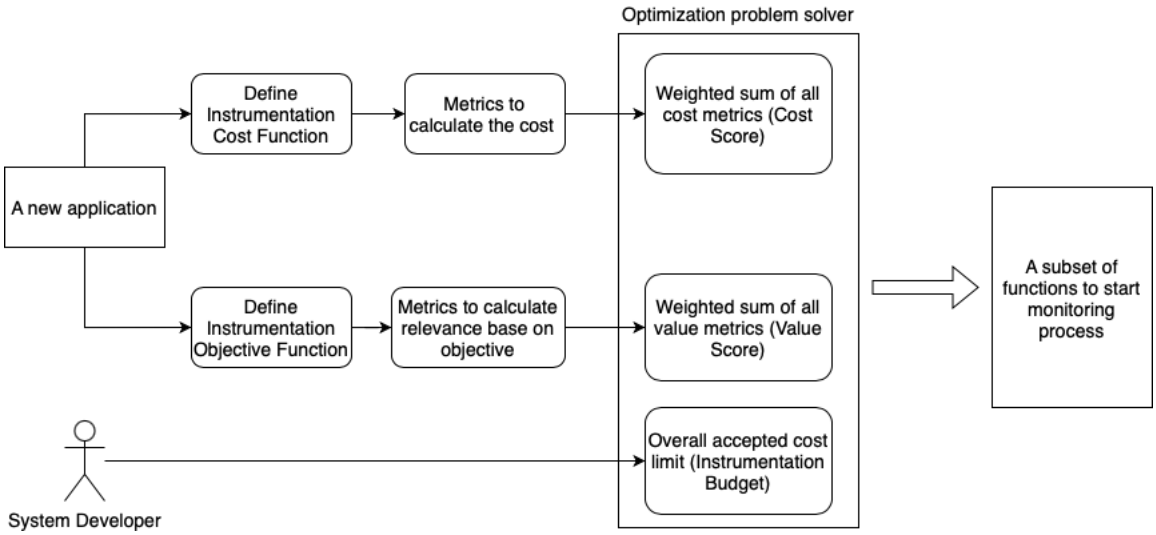

Figure 4.2 Overall Workflow of Cost-Aware Instrumentation framework

overhead, memory consumption, storage requirements, or network bandwidth depending on their operational environment and priorities. Based on their system knowledge and operational constraints, the developer would then establish acceptable limits for each selected cost function, creating the constraint parameters that reflect their specific deployment requirements. The framework would automatically extract static metrics from all Redis functions and apply the appropriate weight factors based on Redis's resource utilization patterns to calculate value scores for each function. Moreover, the cost measurement component would quantify the overhead associated with different instrumentation configurations according to the selected cost functions. The framework would then formulate and solve the knapsack optimization problem using these inputs, automatically identifying the optimal subset of functions that maximizes the specified objective while respecting all cost constraints. Finally, the framework generates an instrumentation configuration that eliminates guesswork from the decision-making process and ensures optimal monitoring coverage within the developer's operational boundaries.

4.5 Methodology

This section presents the methodology we adopted to develop a prototype implementation of the framework formulated in Section 4.3. The prototype serves to demonstrate the practical feasibility of our systematic optimization approach and validate the effectiveness of the mathematical formulation across diverse application types. Our implementation follows the framework application guidelines presented in Section 4.4, adapting the general principles to

specific technical constraints and evaluation requirements.

Rather than implementing all possible cost dimensions and monitoring objectives identified in our taxonomy, this prototype focuses on a representative subset that enables thorough evaluation while maintaining implementation tractability. This approach allows us to validate the core optimization methodology and demonstrate the ability of our framework to identify near-optimal instrumentation configurations under realistic constraints.

Our methodology systematically implements the mathematical framework through several key steps: defining the instrumentation objective and cost scenarios, developing methods for calculating code region value scores, establishing techniques for measuring instrumentation costs, and implementing optimization algorithms to solve the resulting knapsack problems. The evaluation encompasses three diverse applications—a CPU-intensive SPEC benchmark, the Redis in-memory database, and the Apache HTTPD web server—selected to represent different resource utilization patterns and computational characteristics.

To demonstrate the flexibility and adaptability of the framework to different system constraints, we identified multiple cost scenarios for evaluation:

1. Execution time overhead only: This scenario considers the additional processing time introduced by instrumentation as the sole cost constraint, representing systems where performance latency is the primary concern.

2. Combined execution time and storage costs: This scenario accounts for both the execution time overhead and the size of trace data generated, addressing systems with both computational and storage limitations.

3. Unlimited cost: This scenario removes cost constraints to evaluate the function selection capabilities of the framework when optimized purely for monitoring value, serving as both a validation of our value scoring methodology and a comparison baseline.

By evaluating the framework under these distinct cost scenarios, we demonstrate its adaptability to different system constraints and provide evidence for its effectiveness across various operational requirements. It is valuable to note that the selection of costs is for evaluation purposes and a comprehensive analysis of more costs combined requires a more targeted analysis. The following subsections detail each implementation step, providing complete methodology for implementing instrumentation frameworks and concrete guidance for re-

searchers seeking to replicate or extend our approach.

Throughout the subsequent sections outlining our prototype implementation and experimental setup, we employ the term *tracing* in substitution with *instrumentation*, to maintain conceptual clarity.

4.5.1 Calculating code region value

Overview

The calculation of code region value represents the cornerstone of our framework, directly implementing the value score calculation (r_k) from our mathematical formulation presented in Section 4.3. This component determines which functions are most likely to exhibit execution time unpredictability and therefore warrant monitoring attention within our optimization-based approach. Rather than relying on ad-hoc developer intuition, our framework requires systematic quantification of function importance to enable automated instrumentation placement decisions that balance monitoring effectiveness against cost constraints.

To achieve this systematic quantification, we employ a metric-based methodology that predicts execution time variability through observable code characteristics. This approach aligns with the requirement of our framework for automated decision-making by transforming subjective assessments of function importance into objective, measurable scores that can feed directly into our knapsack optimization formulation.

Our methodology is based on established research in performance prediction and software quality assessment. Laaber et al. [124] developed static metrics for predicting unstable software benchmarks, while Majd et al. [125] introduced metrics for pinpointing statement-level defects based on code characteristics. These studies demonstrate that observable code features can effectively predict runtime behavior variability. We adopt similar metrics but adapt the approach specifically for cost-aware instrumentation decisions, where the goal is identifying functions whose monitoring provides maximum diagnostic value relative to instrumentation overhead.

The specific characteristics that we need to predict for effective cost-aware instrumentation are functions that exhibit inconsistent execution times across different operational conditions. Such functions often indicate performance bottlenecks, resource contention, or state-

dependent behavior that requires monitoring attention. To capture these characteristics systematically, we extract metrics that reflect both computational complexity and resource interaction patterns, as these factors most strongly influence execution time variability.

Building on the comprehensive mapping study by Nunez-Varela et al. [126] on static metrics across programming languages, we focus specifically on metrics with direct execution time impact. The metrics we selected, illustrated in Table 4.2, are categorized into two strategic groups that align with different sources of execution time variability. On-CPU metrics capture computational characteristics including loops, branches, and recursion that influence execution time fluctuation during active processing. Off-CPU metrics capture resource interaction patterns such as file access and lock access frequency that induce execution time variations through wait states, resource contention, or external dependencies.

To establish the relationship between these observable metrics and actual execution time unpredictability, we conducted a systematic experiment measuring function behavior under diverse operational conditions. Since no existing datasets provide the combination of code metrics and execution time variability measurements required for our cost-aware framework, this experimental validation was essential to derive reliable weight factors for our optimization process.

A critical requirement for the practical applicability of our framework is adaptability to different application domains, as the relative importance of various metrics differs significantly based on resource utilization patterns. CPU-intensive applications exhibit different predictive relationships than IO-intensive or network-intensive applications, directly connecting to the diverse cost profiles identified in our cost taxonomy from Section 4.3. Consequently, we derived separate weight factors for four application categories: CPU-intensive, IO-intensive, memory-intensive, and network-intensive applications, as presented in Table 4.2.

Recognizing that real-world applications rarely utilize a single resource exclusively, our framework accommodates mixed-resource scenarios through a resource-proportional weighting approach. For applications spanning multiple categories—such as Redis, which exhibits intensive IO and network activity—we combine weight factors based on actual resource utilization distribution. This adaptive capability ensures that our framework accurately reflects the composite nature of modern applications while maintaining the systematic optimization approach essential for cost-aware instrumentation decisions.

This value calculation methodology enables our framework to automatically assess func-

tion monitoring importance across diverse application domains, providing the systematic foundation required for optimization-based instrumentation placement while supporting the cost-aware decision-making process central to our approach.

Table 4.2 Weight factor for the extracted metrics.

Features	CPU-Intensive	IO-Intensive	Network-Intensive	Memory-Intensive
Number of loops	0.25	0.15	0.10	0.20
Number of nested loops	0.15	0.10	0.05	0.20
Is recursive	0.05	0.05	0.02	0.04
Number of branches	0.20	0.10	0.05	0.12
Number of threads	0.05	0.05	0.02	0.15
Number of forked processes	0.02	0.01	0.01	0.03
Number of exception handling	0.08	0.04	0.02	0.05
number of lock access	0.10	0.20	0.15	0.15
Number of file access	0.05	0.20	0.25	0.02
Number of input wait	0.03	0.10	0.10	0.01
Number of system call	0.02	0.05	0.10	0.02
Number of socket access	0.05	0.10	0.25	0.01

Implementation Detail

To implement the metric-based value calculation methodology described in the overview, we developed an automated analysis pipeline that extracts both static and dynamic features from applications and calculates the weight factors required for our optimization framework. This implementation transforms the theoretical approach into a practical system capable of analyzing different types of application and generating the quantitative input needed for our cost-aware instrumentation decisions. The overall flow of our implementation detail is shown in Figure 4.3

As we can see from the figure, for the extraction of static metric from the source code of the application, we used srcML[2], an open source tool that facilitates static analysis for multiple programming languages including C, C++, C#, and Java. This tool proved suitable for our metric extraction requirements, enabling automated parsing of all source files in a directory and extraction of the metrics outlined in Table 4.2 from each function. The automated parser processes the entire code base and generates structured output containing metric values for

[2]https://www.srcml.org

each function, eliminating manual analysis effort while ensuring consistent metric extraction across different applications.

The calculation of reliable weight factors required evaluation using applications that represent single resource utilization patterns. We selected four distinct single-resource benchmarks to isolate the impact of individual resource types on execution time variability. For CPU-intensive applications, we employed a SPEC CPU benchmark that focuses exclusively on computational processing [3]. For IO-intensive evaluation, we utilized the Dbench [4] benchmark, which generates filesystem load by mimicking IO calls from the smbd server, without involving networking operations. For memory-intensive analysis, we selected LMBench[5], a memory and operating system benchmark suite containing multiple memory subsystem tests that examine bandwidth, latency, and access patterns. For network-intensive evaluation, we employed the tbench[6] benchmark, which generates TCP and process loads by replicating socket calls under netbench conditions while preventing file system interference.

This selection of single-resource benchmarks enables clean weight factor calculation without interference from mixed resource usage, ensuring that each weight factor accurately represents the relationship between metrics and execution time variability for its specific resource category. The focus on single-resource applications during weight factor calculation provides the foundation for accurately handling mixed-resource scenarios in real-world applications.

Dynamic metric collection required performance testing, to capture function behavior under different operational conditions that simulate real-world cost constraint scenarios. We employed uftrace[7], a function call graph tracer capable of monitoring programs written in C, C++, Rust, and Python, to collect execution time information during each performance test. The performance testing encompassed five distinct operational conditions designed to stress different system resources: running applications with varied input data under normal conditions, imposing heavy CPU load to intensively utilize processing capacity, saturating memory to maximum capacity, subjecting systems to intensive I/O operations, and adding heavy network operations. These conditions simulate the different resource stress scenarios that applications encounter in production environments, enabling validation that our metrics predict variability across the different cost dimensions identified in our framework.

[3]https://www.spec.org/cpu2017/Docs/overview.html#benchmarks
[4]https://www.samba.org/ftp/unpacked/dbench/README
[5]https://github.com/intel/lmbench
[6]https://www.samba.org/ftp/unpacked/dbench/README
[7]https://github.com/namhyung/uftrace

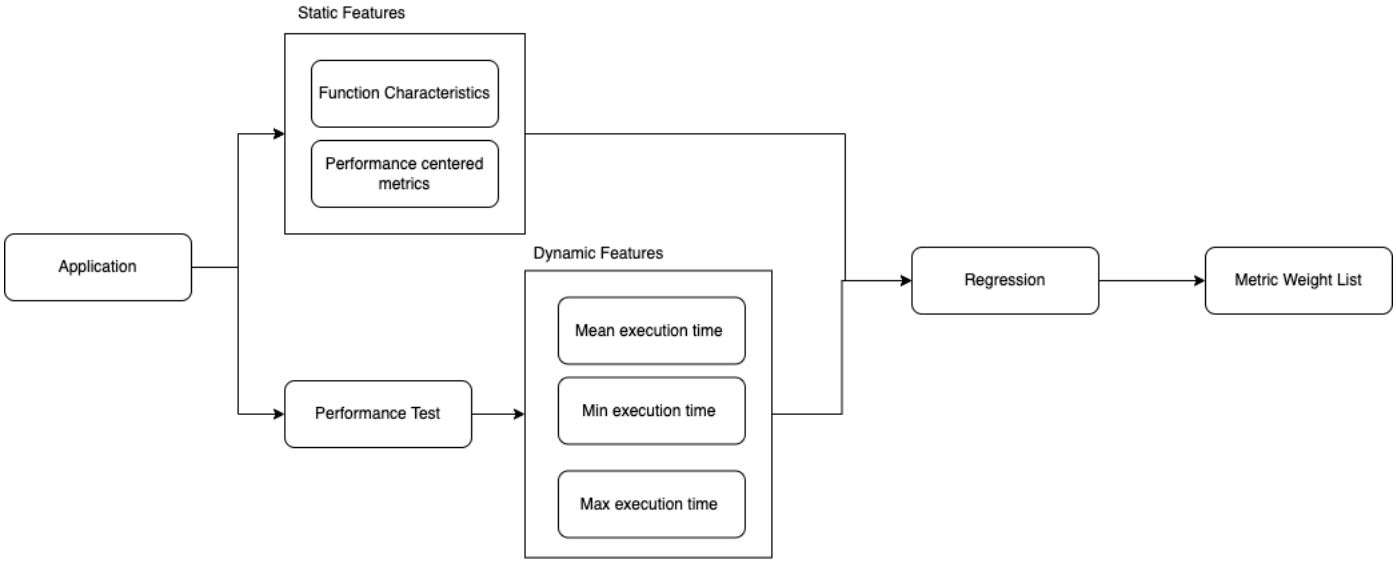

Figure 4.3 Workflow of calculating the weight factor of metrics

To ensure function behavior coverage during performance testing, we leveraged large language models to generate additional input files of varying sizes and characteristics while preserving the semantic structure of original benchmark inputs. This approach expanded our testing beyond standard benchmark inputs, providing broader coverage of realistic execution scenarios and higher confidence in the performance measurements used for weight factor calculation.

To verify that our metrics framework applies to different applications with similar resource utilization profiles, we conducted systematic validation using independent verification benchmarks. We selected the following benchmarks: for CPU-intensive application, we chose CoreMark[8], we selected Filebench[9] as an I/O-intensive benchmark and, and iperf3, which is a network-intensive application[10].

Our validation process followed four systematic steps to ensure the reliability and generalizability of our weight factors. First, we extracted the same static metrics from these verification benchmarks as we did for our primary benchmarks, using the identical automated parsing approach to maintain consistency in metric collection. Second, we applied our established weight factors to calculate predicted variability scores for functions in the verification benchmarks, using the corresponding single-resource weight factors for CoreMark (CPU-intensive), Filebench (IO-intensive), and iperf3 (network-intensive).

[8]https://github.com/eembc/coremark
[9]https://github.com/filebench/filebench
[10]https://github.com/esnet/iperf

Third, we performed performance tests on these benchmarks under different operational conditions to measure actual execution time variability, replicating the same five stress conditions used in our primary experiments. For each function in the verification benchmarks, we collected execution time measurements across multiple runs and calculated the coefficient of variation to quantify actual execution time unpredictability. Finally, we calculated the correlation between predicted and actual variability values to assess the accuracy of our metric-based predictions, providing quantitative evidence of how well the predictions of our framework align with observed function behavior in independent applications.

Most real-world applications rarely utilize a single system resource exclusively, but rather operate across multiple resource domains simultaneously. To accurately calculate weight factors for these mixed-resource applications, we propose a resource-proportional approach that employs a weighted mean based on actual resource utilization patterns. This approach requires minimal profiling effort—a single execution of the application under a representative workload is sufficient to determine the distribution of resource utilization. For instance, if profiling reveals that an application spends 70% of its execution time utilizing CPU resources and 30% performing I/O operations, the composite weight factor would be calculated as 0.7 times the CPU-intensive weight factor plus 0.3 times the I/O-intensive weight factor. This proportional mean creates customized weight factors that more accurately reflect the actual resource utilization profile of the application, thereby improving the precision of execution time variability predictions.

Results

The results of the metric weight calculation step are presented in Table 4.2, which provides an overview of the weight factors for the extracted metrics. A detailed analysis of the table reveals that each metric exhibits distinct values depending on the application domain. Notably, the number of loops demonstrates a higher weight factor in CPU-intensive applications compared to the other application types. Similarly, network-intensive applications exhibit the highest weight for the number of socket accesses metric, while IO-intensive applications display significant weight factors for the number of lock accesses and file accesses. For memory-intensive applications, loops, in addition to the number of threads, have the highest impact compared to other metrics. It is important to note that for applications that do not strictly belong to a single domain, we can derive the weight factor by averaging the

most influential metrics from each relevant aspect.

Table 4.3 Result of Weight Factor Validation Process.

Verification Benchmark	Resource Type	Related Primary Benchmark	Correlation	p-Value
CoreMark	CPU-Intensive	SPEC CPU	0.873	0.000197
FileBench	IO-Intensive	Dbench	0.792	0.000346
iPerf3	Network-Intensive	Tbench	0.731	0.000512

The results of validating the extracted weight factors are shown in Table 4.3. For each of the validation benchmarks, we used the weight factors that accurately represent their resource utilization. We can observe from the table that there is a relatively high correlation between the predicted CoV and the actual CoV extracted after the performance test, with correlation values of 0.873 for CoreMark, 0.792 for FileBench, and 0.731 for iPerf3. We also calculated the p-value to ensure that the correlation is statistically significant, with all values well below the standard significance threshold. These results demonstrate that our framework generalizes across different applications within each resource category, indicating that the derived weight factors can be applied to other applications with similar resource utilization patterns. The validation confirms that our metric-based approach predicts execution time variability, enabling practical deployment of the framework for automated function selection in real-world instrumentation scenarios.

4.5.2 Calculating The Instrumentation Cost

Overview

The calculation of instrumentation cost represents the second essential component of our framework, directly implementing the cost measurement (c_k) from our mathematical formulation presented in Section 4.3. Building upon our cost taxonomy, we selected two representative cost types that demonstrate the capability of our framework to handle diverse overhead categories across multiple dimensions while enabling systematic quantification for optimization-based instrumentation decisions.

From our multi-dimensional cost categorization, we selected execution time overhead and storage requirements to validate the practical cost measurement capabilities of our framework. Execution time overhead represents intrinsic/operational/immediate costs, capturing

the computational burden that manifests directly during instrumentation execution. Storage requirements represent intrinsic/operational/deferred costs, reflecting the persistent overhead that accumulates over time through trace data generation. This selection demonstrates the ability our framework to handle both dynamic costs that require runtime measurement and predictable costs that can be quantified through static analysis approaches.

These two cost types illustrate fundamentally different measurement challenges that validate the systematic approach of the framework to cost quantification. Execution time overhead requires precise runtime measurement techniques to capture the dynamic performance impact during application execution, while storage costs can be systematically predicted through analysis of trace data generation patterns and payload characteristics. By successfully measuring both immediate runtime impacts and longer-term storage implications, our framework demonstrates the capability to handle the diverse cost spectrum identified in our taxonomy while providing the complete cost input set required for the knapsack optimization formulation.

The systematic measurement of these costs enables our optimization framework to make data-driven instrumentation decisions that replace the ad-hoc developer approaches identified as problematic in Section I. By quantifying both immediate runtime overhead and persistent storage costs, the framework can optimize instrumentation placement for different operational priorities and resource constraints, providing the precise cost scores (wk) necessary for our mathematical optimization process and supporting informed trade-offs between monitoring effectiveness and system overhead.

Implementation Detail

Conventional overhead measurement approaches typically compare application execution with and without instrumentation enabled, yielding only aggregate cost information across the entire system. This approach fails to provide the function-level cost granularity required by our optimization framework, where the knapsack formulation demands individual cost scores for each potential instrumentation point. To address this fundamental limitation and enable the precise cost-to-function mapping essential for systematic optimization, we developed a micro-benchmark methodology that isolates and quantifies instrumentation overhead at the individual function level.

Our approach employs LTTng (Linux Trace Toolkit: next generation) as the measurement

platform due to its capability to provide controlled, repeatable cost measurements for individual trace points across different execution contexts. The user-space tracing capabilities of LTTng enable precise timing measurement and payload size quantification, directly supporting the requirement of our framework for systematic cost calculation across the immediate (execution time) and deferred (storage) cost categories identified in our taxonomy. This tool selection enables validation that our cost measurement approach can handle both dynamic runtime costs and predictable storage costs within a unified measurement framework.

The micro-benchmark methodology systematically captures cost information through controlled experimentation designed to validate the multi-dimensional cost handling capability of our framework. We define custom trace points that represent typical instrumentation scenarios, then measure both execution time overhead and storage requirements across varying payload sizes and system conditions. This approach generates the precise cost mappings necessary for our optimization formulation, enabling prediction of instrumentation overhead for any function based on its input/output characteristics and expected call frequency.

To ensure measurement accuracy and statistical significance, our implementation incorporates a warm-up phase of 50 iterations to eliminate cold-start effects, followed by 1,000 measurement iterations for each trace point configuration. This methodology provides the reliable cost data required by our optimization framework while demonstrating that systematic cost measurement is computationally feasible across the diverse instrumentation scenarios encountered in real-world applications. The resulting cost mappings enable our framework to predict instrumentation overhead without requiring runtime measurement during the optimization process, supporting efficient deployment of cost-aware instrumentation decisions.

Through extensive testing across different payload sizes (4 bytes to 2 kilobytes) and system conditions, we validate that our cost measurement approach successfully captures the cost variability essential for accurate optimization. This systematic cost quantification transforms the previously ad-hoc cost estimation process into a data-driven component of our optimization framework, providing the precise cost inputs necessary for the knapsack formulation while demonstrating the practical feasibility of systematic cost-aware instrumentation.

Results

The micro-benchmark evaluation successfully demonstrates the feasibility of systematic cost measurement required by our optimization framework, providing precise cost mappings that enable the wk calculations essential for knapsack optimization. Figure 4.4 presents the execu-

tion time cost results, revealing a clear polynomial relationship between trace point payload size and execution overhead that ranges from approximately 200 nanoseconds for minimal payloads to over 1,400 nanoseconds for 2-kilobyte payloads. This relationship enables our framework to predict instrumentation execution costs for any function based on its input/output data characteristics, directly supporting the systematic cost quantification required by our mathematical formulation.

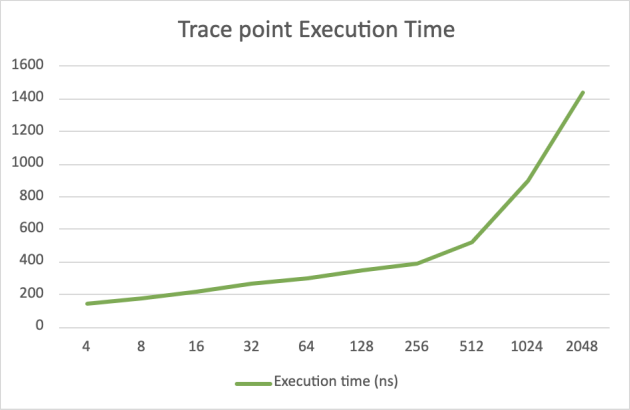

Figure 4.4 Result of trace point execution time benchmark

The cost-payload relationship validates an assumption of our framework: that instrumentation costs can be systematically predicted and quantified rather than estimated through ad-hoc approaches. The polynomial growth pattern indicates that payload size significantly impacts execution overhead, enabling our optimization framework to make informed trade-offs between monitoring effectiveness and performance impact. Functions with large input/output parameters incur proportionally higher costs, allowing the knapsack optimization to automatically balance monitoring value against increased overhead when selecting instrumentation points.

For storage cost calculation, our analysis demonstrates that trace data requirements follow a predictable pattern combining fixed baseline costs with variable payload-dependent components. Each trace point generates baseline metadata including process ID, thread ID, timestamp, and event type information (approximately 32 bytes), while custom payload data scales linearly with function parameter sizes. For example, a function with two integer parameters generates 40 bytes per invocation (32 bytes baseline + 8 bytes payload), while complex data structures proportionally increase storage requirements. When multiplied by expected function call frequencies, this approach provides precise storage cost predictions that integrate with execution time costs in our multi-constraint optimization formulation.

The validation of our cost measurement approach against actual instrumentation scenarios confirms that micro-benchmark results represent real-world overhead patterns. Testing across different system conditions and payload variations provides the reliability necessary for optimization-based decision making. This consistency validates that our systematic cost measurement approach successfully captures the cost variability essential for accurate instrumentation placement, transforming previously unpredictable overhead estimation into data-driven cost prediction.

These results enable the complete cost score calculation (w_k) required by our knapsack formulation, where the instrumentation cost of each function can be predicted by combining its execution time overhead with its storage requirements. This systematic cost quantification provides the precise input necessary for the optimization problem solving described in the subsequent section, demonstrating that cost-aware instrumentation decisions can be automated through mathematical optimization rather than relying on developer intuition. The cost mapping established through this methodology forms the foundation for systematic instrumentation placement that respects specified resource constraints while maximizing monitoring effectiveness.

4.5.3 Solving Optimization Problem

Overview

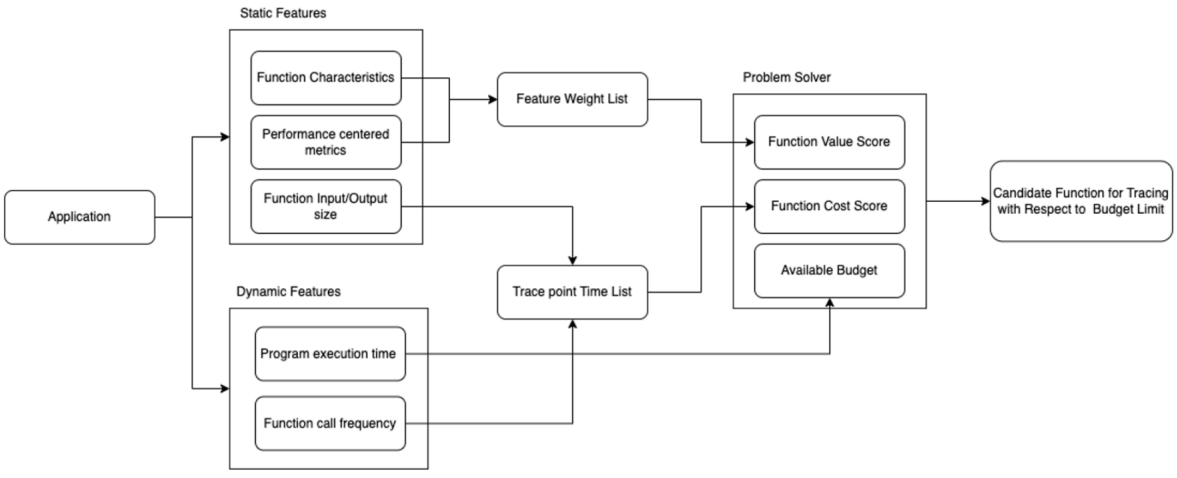

Figure 4.5 Implementation of Cost-Aware Instrumentation Framework

This final component integrates the value scores (r_k) and cost scores (w_k) to implement the complete optimization framework formulated in Section 4.3. The optimization process transforms the theoretical cost-aware instrumentation problem into actionable function selection recommendations by systematically balancing monitoring effectiveness against resource constraints. Figure 4.5 illustrates this integration, demonstrating how cost analysis, objective specification, and optimization components combine to produce systematic instrumentation decisions that replace the ad-hoc developer approaches identified as problematic in Section 4.1.

The optimization implementation validates our multidimensional cost taxonomy by successfully handling diverse cost types within a unified mathematical framework, and the optimization process respects budget constraints across different resource dimensions while maximizing monitoring value. This approach enables developers to specify their operational priorities and resource limitations, with the framework automatically identifying instrumentation configurations that respect these constraints while providing optimal diagnostic coverage.

The selection of algorithms to solve the resulting knapsack optimization aligns with the requirements of the framework for both the quality of the solution and the computational tractability. Dynamic programming algorithms provide guaranteed optimal solutions for single-constraint scenarios and smaller applications, ensuring maximum monitoring effectiveness within specified budgets. For large-scale applications or scenarios that require rapid deployment, genetic algorithms offer near-optimal efficient solutions with acceptable computational overhead. This algorithmic flexibility ensures that the framework can adapt to different deployment contexts while maintaining optimization principles, providing practitioners with instrumentation recommendations that significantly outperform random or intuition-based selection approaches.

The optimization component serves as a validation of our entire cost-aware framework, demonstrating that systematic instrumentation placement can be automated through mathematical optimization while respecting the complex cost relationships identified in our taxonomy. By successfully integrating value prediction, cost measurement, and constraint optimization, this component enables practical deployment of cost-aware instrumentation across diverse application domains and operational requirements.

Table 4.4 Program characteristics used for evaluation phase

Program	Domain	Function Count	Duration No Tracing	Duration Full Tracing
Data compression (SPEC CPU)	CPU-Intensive	367	4m 6s	24m 56s
Redis Database	Network/memory/IO Intensive	4153	2m 35s	11m 51s
HTTPD Web Server	Network-Intensive	6396	50s	4m 23s

Implementation Detail

To validate our framework across diverse operational constraints, we designed three evaluation scenarios that demonstrate the optimization capabilities of the framework. Each scenario validates a distinct aspect of our theoretical contributions: single-cost constraint optimization, multi-dimensional cost handling, and pure value-based selection. This evaluation approach enables the assessment of how our optimization framework performs under different resource limitations while demonstrating consistent improvement over ad-hoc instrumentation approaches.

Our application selection represents different resource utilization patterns identified in our cost taxonomy, enabling validation across diverse computational characteristics. The SPEC CPU benchmark represents CPU-intensive workloads with primarily computational overhead, Redis demonstrates multi-resource applications combining network, memory, and I/O operations, and Apache HTTPD exemplifies network-intensive systems with communication-dominated costs. These applications collectively validate that our framework adapts effectively to different resource profiles while maintaining optimization effectiveness, as outlined in Table 4.4.

The first evaluation scenario validates single-cost constraint optimization using the SPEC CPU benchmark under a 5% execution time overhead budget. This constraint reflects realistic production environments where performance degradation must be minimized while maintaining effective monitoring coverage. By limiting overhead to 5% of baseline execution time, we demonstrate the ability of our framework to identify optimal instrumentation configurations within tight resource constraints typical of latency-sensitive applications.

The second scenario validates multi-dimensional cost handling through Redis evaluation under dual constraints: 10% execution time overhead and 200MB storage limitation. These constraints reflect real-world scenarios where applications face simultaneous resource limitations across different cost dimensions identified in our taxonomy. The 10% execution time budget represents acceptable performance impact for database systems, while the storage

constraint simulates operational environments with limited disk capacity or network bandwidth for trace data transmission.

The third scenario validates pure value-based function selection using Apache HTTPD under unconstrained cost conditions. By removing cost limitations and selecting the top 10% of functions based solely on monitoring value. This scenario demonstrates the function ranking capabilities of our framework when optimized exclusively for diagnostic effectiveness. This unconstrained evaluation serves as both validation of our value scoring methodology and baseline comparison to understand how cost constraints influence instrumentation decisions.

For multi-constraint scenarios, we implement the weighted normalization approach established in Section 4.3 to transform the multi-dimensional optimization problem into a tractable single-constraint formulation. Each cost type is first normalized against its respective budget to ensure comparability across different resource dimensions. The weighting factors for combining normalized costs are determined using the Average Normalized Cost Method, which reflects the relative constraint tightness of each resource type by calculating the average normalized cost across all functions for each cost dimension. This approach ensures that more constraining resources receive proportionally higher influence in optimization decisions, enabling systematic handling of competing resource limitations while preserving the mathematical tractability of the optimization framework.

Through this systematic evaluation design, we validate that our optimization framework consistently outperforms random or intuition-based instrumentation selection across varying constraint conditions. The progressive scenario structure—from single-cost through multi-cost to unconstrained optimization—demonstrates the adaptability and effectiveness of our framework across the complete spectrum of operational requirements, providing evidence for the advantages of systematic cost-aware instrumentation over traditional ad-hoc approaches.

Results

Table 4.5 Result of evaluating a prototype of the proposed framework

Program	Available Instrumentation Cost	Selected Function Count	Description
Data compression (SPEC CPU)	5% of Execution Time	53	Selected from top 27%
Redis Database	10% of execution time / 200MB trace data	314	Selected from top 12%
HTTPD web server	Unlimited budget	639	Selected from top 17%

The evaluation results provide a validation of our complete cost-aware instrumentation framework, demonstrating that systematic optimization can identify functions with execution time unpredictability across diverse application domains and constraint scenarios. We employ the Coefficient of Variation (CoV) as our validation metric because it directly measures execution time unpredictability, the fundamental characteristic that our framework aims at for effective performance monitoring. Functions with high CoV values exhibit inconsistent execution times across different runs and system conditions, often indicating performance bottlenecks, resource contention, or state-dependent behavior that requires monitoring attention. This validation approach enables quantitative assessment of how well our optimization framework identifies critical instrumentation points compared to random or ad-hoc selection approaches.

The systematic evaluation across three scenarios reveals the optimization precision, validating both our theoretical framework and its practical implementation feasibility. In the CPU-intensive scenario (SPEC CPU benchmark) with 5% execution time constraint, our framework identified 53 functions that ranked within the top 27% based on actual CoV measurements. This represents an improvement over random selection, which would achieve an average ranking of approximately 50%. The multi-resource scenario (Redis) with dual constraints (10% execution time, 200MB storage) demonstrated capability of selecting 297 functions from the top 12% of CoV rankings. This validates that our multi-dimensional cost handling and weighted normalization approach successfully eliminates functions with poor cost-to-value ratios, concentrating selection on functions providing maximum monitoring insights per unit cost incurred.

The unconstrained scenario (Apache HTTPD) selecting the top 10% of functions based purely on monitoring value achieved 18% CoV ranking precision, validating our value scoring methodology while providing baseline comparison for cost-constrained optimization. The correlation between predicted function importance and actual execution time variability across all scenarios—spanning CPU-intensive, multi-resource, and network-intensive applications—validates that our combination of static and dynamic metrics successfully predicts runtime behavior patterns, confirming the soundness of our optimization approach for identifying critical instrumentation points.

Table 4.5 summarizes these validation results, demonstrating that our framework implementation successfully handles diverse application characteristics, constraint type, and resource utilization patterns, while maintaining consistent optimization effectiveness. The results

validate our theoretical contributions including multi-dimensional cost quantification, systematic value scoring, and optimization-based placement decisions, providing evidence that cost-aware instrumentation can be automated through mathematical optimization rather than relying on developer intuition.

The prototype implementation successfully demonstrates the practical feasibility of deploying our framework across diverse application domains and operational constraints. By consistently identifying high-variability functions while respecting specified resource limitations, this implementation validates that systematic cost-aware approaches can effectively replace ad-hoc instrumentation practices, providing developers with principled tools for making informed monitoring decisions that balance diagnostic effectiveness against system overhead.

4.6 Threats To Validity

This section discusses the threats to the validity of our research study, organized according to established validity categories. For each threat, we discuss both the limitations and the mitigation strategies we employed to address them.

4.6.1 Construct Validity

Construct validity concerns whether our measurements actually capture the intended concepts. Our framework uses execution time variability, measured through the Coefficient of Variation (CoV), as a proxy for functions that warrant performance monitoring attention. While CoV effectively captures execution time unpredictability, it represents only one aspect of performance issues. Functions with consistent execution times may still be critical for monitoring if they consume significant resources or represent bottlenecks under specific conditions.

To address this limitation, we selected CoV specifically because it directly measures the execution time unpredictability that our framework targets for performance monitoring. We validated our approach by demonstrating consistent correlation between our predicted function rankings and actual variability measurements across different application types. Additionally, our static metrics selection was based on established research by Laaber et al. and Majd et al., who demonstrated similar metrics' effectiveness in predicting unstable software behavior and performance-related defects.

4.6.2 Internal Validity

Selection bias may exist in our study regarding both the applications chosen for evaluation and the benchmarks used for weight factor extraction. To mitigate this bias, we deliberately selected applications representing different resource utilization patterns: CPU-intensive (SPEC CPU), multi-resource (Redis), and network-intensive (Apache HTTPD). For weight factor calculation, we chose benchmarks that focus exclusively on single resource types (SPEC CPU for CPU, Dbench for I/O, LMBench for memory, tbench for network) to ensure clean separation of resource-specific characteristics.

Our performance tests conducted for weight factor calculation could introduce bias due to potential oversight of certain application execution paths. To address this risk, we incorporated diverse workload scenarios including varied input data, CPU stress, memory saturation, intensive I/O operations, and heavy network load. We also leveraged Large Language Models to systematically generate additional input files of varying sizes and characteristics while preserving semantic structure, enabling broader coverage of realistic execution scenarios than would be possible with standard benchmark inputs alone.

The accuracy of our metric extraction depends on tool capabilities. We employed state-of-the-art tools specifically tailored for each phase: srcML for static analysis (supporting multiple programming languages), and uftrace for dynamic metric collection (providing precise function-level tracing). We validated our extraction accuracy through cross-verification with multiple profiling approaches.

4.6.3 External Validity

The implementation and evaluation of our framework are specific to the C programming language, which limits generalizability to other programming languages. We acknowledge this limitation but chose C because it represents widely-used systems programming languages where performance monitoring is critical. To demonstrate broader applicability, we selected diverse applications within the C domain that represent different computational patterns and resource utilization profiles commonly found across software systems.

Our evaluation encompasses three applications, which may not fully represent all software domains. However, we deliberately chose applications that span different categories: a stan-

dardized benchmark (SPEC CPU) for reproducibility, a widely-deployed database system (Redis) for real-world relevance, and a foundational web server (Apache HTTPD) for network-intensive workloads. This selection strategy provides coverage across primary resource utilization patterns while maintaining evaluation tractability.

Our cost analysis focuses primarily on execution time overhead and storage costs, while acknowledging that real-world deployments may face additional constraints. We selected these cost types because they represent the most commonly cited concerns in instrumentation literature and can be measured accurately across different systems. The mathematical formulation of the framework is designed to accommodate additional cost types, and our multi-constraint optimization approach provides a foundation for incorporating other cost dimensions as they become measurable and relevant.

4.7 Discussion

Our proposed framework represents an advancement in systematic instrumentation research through its multi-dimensional cost taxonomy and optimization-based placement methodology. The primary contribution of this work lies in establishing the first systematic categorization of instrumentation costs across multiple dimensions, providing researchers and practitioners with a structured foundation for understanding the full spectrum of overhead types and their relationships. This framework addresses the gap in existing literature, where cost considerations have been addressed in fragmented, single-dimension approaches.

Although this study focused on CPU and memory utilization due to their universal importance in performance engineering, the proposed two-phase feature selection and modeling pipeline is not inherently tied to these metrics. The same methodology can be applied to other resource dimensions, such as disk I/O, network bandwidth, RPC latency, or storage throughput, provided that appropriate ground-truth measurements are available.

The evaluation results demonstrate the practical viability of our optimization-based approach for identifying critical functions across diverse application domains. Our framework identified functions that ranked within the top 12-27% based on actual execution time variability measurements. This precision outperforms random selection approaches and provides evidence that systematic optimization can effectively balance monitoring value against cost constraints. The ability of the framework to maintain effectiveness across different resource utilization profiles—CPU-intensive (SPEC CPU), network-intensive (Apache HTTPD), and

multi-resource applications (Redis)—validates both the adaptability of our resource-specific weight factors and the robustness of our optimization algorithm.

A critical aspect of our work involves positioning it within the broader landscape of instrumentation placement research. The two most closely related studies, Log20 [23] and Log4Perf [94], share some conceptual similarities but differ significantly in scope and approach. Log4Perf focuses exclusively on web-based client-server applications and optimizes logging placement for HTTP request performance analysis through statistical analysis of communication patterns. However, this approach neither incorporates cost considerations into the optimization process nor generalizes beyond web-based systems, making direct comparison impractical due to fundamental differences in problem formulation.

Log20 presents a more comparable approach by formulating instrumentation placement as a cost-aware optimization problem that maximizes informativeness while respecting execution time overhead constraints. Our framework extends this concept significantly by providing systematic multi-dimensional cost analysis rather than focusing solely on execution time overhead. While direct experimental comparison was not feasible due to the unavailability of the implementation artifacts for Log20, our approach distinguishes itself through cost taxonomy, multi-constraint optimization capabilities, and systematic methodology for diverse application domains. This limitation in tool availability highlights a broader reproducibility challenge in instrumentation research that future work should address.

Our prototype demonstrates the effectiveness of function-level tracing through static code analysis, establishing a foundation for cost-aware instrumentation in a wide range of application scenarios. The framework excels in environments where source code is accessible, making it valuable for in-house developed applications, open-source systems, and enterprise software where complete code visibility enables comprehensive analysis. Applications with stable execution patterns and predictable resource utilization characteristics—such as database systems, web servers, and data processing applications—benefit from our approach, as their computational behavior aligns well with our metric-based prediction model. The framework shows performance for monolithic applications and well-defined service boundaries where our established resource utilization categories provide characterization. However, our approach has limitations that define its applicable scope. Applications with high workload variability require frequent weight factor recalibration, which may not be practical in all operational environments. Microservices architectures present challenges, as the framework requires complete source code access for analysis, making it unsuitable for scenarios involving third-party

services or independently developed components. Additionally, applications where runtime behavior diverges from static code characteristics may experience reduced prediction accuracy. These limitations represent constraints of the static analysis approach and highlight scenarios where alternative instrumentation strategies may be more appropriate.

The practical deployment characteristics of the framework warrant careful consideration. While our implementation reduces manual effort compared to traditional ad-hoc approaches, initial setup requires systematic profiling to determine application resource utilization patterns and extract dynamic features. For new applications, this involves a single execution under representative workload conditions to collect multiple required inputs: resource utilization distribution for weight factor calculation, baseline execution metrics for cost budget establishment, and function call patterns for cost estimation. When application code undergoes significant changes, additional profiling enables framework reconfiguration and generation of updated instrumentation recommendations. This level of setup effort represents a reasonable investment for applications requiring ongoing performance monitoring, though it may be excessive for short-term or experimental projects.

Our evaluation, while demonstrating consistent technical effectiveness, is constrained by several factors that warrant acknowledgment. The focus on three applications, though strategically selected to represent diverse computational patterns, limits the breadth of evidence for generalizability claims. Additionally, our validation approach measures technical correctness of function selection—correlation between predicted importance and actual execution time variability—rather than practical effectiveness in real-world debugging scenarios. While this technical validation provides necessary evidence for our optimization approach, future work must demonstrate diagnostic utility in actual performance troubleshooting contexts.

The current implementation of the framework addresses a subset of the cost categories identified in our taxonomy. While we demonstrate multi-constraint optimization with execution time and storage costs, the full realization of our cost-aware approach would require systematic implementation across additional cost dimensions such as energy consumption, security overhead, and maintenance burden. Each cost type presents unique measurement challenges and would require dedicated research effort to develop reliable quantification methods.

Despite these limitations, our work establishes important foundations for systematic cost-aware instrumentation research. The cost taxonomy provides a structured framework for future investigations, while our mathematical optimization formulation offers a principled

approach to instrumentation placement decisions. The demonstration of consistent technical effectiveness across diverse application types suggests that optimization-based approaches can provide systematic alternatives to current ad-hoc practices.

The broader implications of this work extend beyond immediate technical contributions. By establishing systematic cost analysis and demonstrating automated placement optimization, our research opens pathways for integrating cost-aware instrumentation into development toolchains and continuous monitoring systems. The mathematical foundation of the framework enables extension to additional monitoring objectives beyond performance analysis, such as security monitoring or fault diagnosis, providing a foundation for instrumentation research.

4.8 Conclusion and Future Work

This research addresses the critical challenge of systematic instrumentation placement in software performance monitoring by introducing a cost-aware framework that transforms ad-hoc developer decisions into principled optimization problems. Traditional instrumentation practices suffer from the lack of systematic approaches to balance monitoring effectiveness against the diverse overhead types introduced by tracing and logging, leading to either excessive performance degradation or insufficient monitoring coverage.

Our work makes several concrete contributions to the field of cost-aware instrumentation. First, we provide the first multi-dimensional categorization of instrumentation costs, establishing a systematic taxonomy that considers intrinsic versus extrinsic, operational versus non-operational, and immediate versus deferred cost types. This categorization addresses a critical gap in existing literature, where cost considerations have been addressed in fragmented, single-dimension approaches. Second, we present a mathematical formulation that transforms the instrumentation placement problem into a tractable optimization framework, enabling automated identification of optimal code sections based on multiple cost constraints simultaneously. Third, our prototype implementation demonstrates the practical feasibility of systematic optimization, consistently identifying functions that ranked within the top 12-27% based on actual execution time variability across diverse application types.

The evaluation of our framework across three representative applications—a CPU-intensive SPEC benchmark, the Redis in-memory database, and the Apache HTTPD web server—provides evidence that optimization-based approaches can effectively replace ad-hoc instrumentation

decisions. The ability of the framework to maintain effectiveness across different resource utilization profiles validates both the adaptability of our metric-based approach and the robustness of our mathematical formulation. By establishing principled cost analysis and automated placement decisions, this work offers a systematic foundation for effective performance monitoring while respecting specified resource boundaries.

While our framework demonstrates promising capabilities, several important research directions emerge from the limitations identified in this study. First, validation across multiple programming languages and broader application domains is essential to establish the generalizability of our approach. The current focus on C applications, while strategically chosen for systems-level performance monitoring, requires extension to other programming paradigms and application types. Second, evaluation in real-world debugging scenarios would strengthen the evidence for practical effectiveness, as our current validation demonstrates technical correctness but not diagnostic utility in actual performance troubleshooting contexts.

Third, systematic implementation across additional cost dimensions identified in our taxonomy represents a substantial research opportunity. While we demonstrate multi-constraint optimization with execution time and storage costs, extending the framework to encompass energy consumption, security overhead, maintenance burden, and other cost types would realize the full potential of cost-aware instrumentation. Each additional cost dimension requires dedicated research to develop reliable quantification methods and understand their interactions with existing cost types.

Fourth, integration with production monitoring systems and development tool chains would enable evaluation of the effectiveness of the framework in continuous monitoring scenarios. This research direction should address dynamic instrumentation adjustment, real-time cost monitoring, and automated reconfiguration as application characteristics evolve. Finally, extension of the optimization framework to additional monitoring objectives beyond performance analysis—such as security monitoring, fault diagnosis, or compliance verification—would demonstrate the broader applicability of systematic cost-aware approaches.

Our framework lays the groundwork for automated, cost-aware observability tools that can adapt to diverse system requirements and operational contexts. The systematic approach to balancing monitoring value against instrumentation overhead opens new possibilities for intelligent observability management in complex computing environments. Future work will focus on extending this framework to dynamic instrumentation scenarios in cloud native platforms and hybrid tracing environments, where distributed system complexity and variable workload

patterns require adaptive monitoring strategies. In addition, we plan to explore real-time adjustment of instrumentation decisions based on changing operational conditions, enabling observability systems that continuously optimize their monitoring coverage in response to evolving system behavior and resource constraints. These extensions will further advance systematic observability management in modern distributed computing environments.

This research establishes essential foundations for systematic cost-aware instrumentation by providing cost analysis, mathematical optimization formulation, and demonstrated technical effectiveness. The framework transforms instrumentation placement from an ad-hoc developer task into a principled optimization problem, offering significant potential for improving software performance monitoring practices. While substantial research remains to fully realize this potential across diverse domains and cost dimensions, our work provides the conceptual and technical groundwork necessary for advancing toward truly systematic instrumentation approaches. To enhance reproducibility and facilitate future research, we have made the source code of our prototype publicly available[11], enabling other researchers to build upon our contributions and extend the capabilities of the framework.

4.9 Acknowledgements

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), AMD, Ciena, Ericsson, and EfficiOS for funding this research project.

[11] https://github.com/Amirhgh74/cost-aware-prototype

CHAPTER 5 ARTICLE 2: AUTOMATIC REDUCTION OF EXECUTION TRACE DATA VOLUME USING GRADIENT BOOSTING IN LARGE-SCALE MICROSERVICE SYSTEMS

Authors: Amir Haghshenas, Naser Ezzati-Jivan, Michel Dagenais

Venue: 37th Canadian Conference on Artificial Intelligence.(Canadian Al 2024)

Submission Date: 18-02-2024

Acceptance Date: 03-04-2024

Abstract In the complex world of online services that rely on interconnected microservices, ensuring these components have the necessary resources for optimal performance is crucial. Traditionally, analyzing extensive logs and traces from these systems has been the method to model performance, but collecting too little or too much data can present challenges for accurate performance modeling. This highlights the need for a more efficient approach. Our study introduces a streamlined two-phase method that leverages gradient boosting algorithms to pinpoint key data features essential for predicting CPU and memory demands accurately. By focusing on feature importance, we were able to significantly reduce the amount of data required for analysis—by more than 69%—without compromising, and in some cases enhancing, the accuracy of our models. This evaluation was significantly strengthened by employing a comprehensive dataset provided by Alibaba, illustrating the practical application and validation of our method in a real-world, large-scale microservice environment. Further analysis on our results reveal that most of the identified features for for data volume reduction were mostly focused on the critical aspects of a microservice architecture, notably inter-service communication and resource access patterns. Our findings demonstrate that by concentrating on the most influential features of the microservice architecture trace data, it is possible to maintain, and potentially improve, system performance modeling with substantially less data, presenting a promising research direction for resource optimization in large-scale microservice performance modeling.

5.1 Introduction

Ensuring continuous operation of microservice architectures is crucial in the technology landscape of recent years, where system availability directly impacts user satisfaction and business operations [33]. One effective strategy to maintain high availability is through detailed performance modeling and resource utilization analysis of the system [127]. By accurately modeling

the performance of the system, organizations can proactively identify potential bottlenecks, optimize resource allocation, and prevent downtime. This approach enhances the reliability of microservices and also supports scalable growth, ensuring systems can handle increasing loads without compromising service quality.

In microservice systems, tracing and logging are crucial for gathering data to analyze system performance and resolve system issues [128]. However, this can create a significant challenge: collecting too much tracing and logging data can lead to sever performance degradation and produce a large amount of unnecessary or irrelevant data. On the other hand, collecting too little data can lead to missing important insights needed for a thorough performance analysis [43]. This situation highlights the need for carefully balanced data collection strategies, aiming to optimize the relevance of the data collected while maintaining system efficiency. In fact, this approach aims to ensure that data collection is both efficient and effective, capturing essential details without cluttering storage with redundant data.

Prior efforts have primarily aimed at optimizing logging through system analysis to determine the ideal quantity and locations for logging, thus ensuring system performance is not affected. Research studies such as Log2 [22], Log20 [23], and Log4Perf [44] have introduced methods that are conscious of costs, using statistical and entropy-based techniques to capture the relevant data and minimize extensive logging. Increasingly, researchers are exploring ways to fine-tune log recommendations using various methodologies such as rule-based [37], heuristic [22, 23], and statistical approaches [129]. The rule-based method focuses on static analysis to identify code patterns critical for logging, though it requires significant expertise and risks missing key patterns. Heuristic methods also use static analysis but face challenges in accommodating multiple objectives within logging decisions. Meanwhile, advances in machine learning and deep learning have shown promise for log placement [130], utilizing techniques such as Decision Trees [105] and LSTM [131]. However, the effectiveness of statistical models can be limited by the challenge of integrating diverse features and adapting to different project logging practices, leading to potential biases in recommendations.

It is clear that while significant progress has been made in log management and recommendation. However, these approaches primarily concentrate on analyzing systems to suggest logging points, a method that may not fully leverage existing trace data for performance modeling. In addition, this approach my not be viable in microservice architecture, since accessing all the available microservice is difficult and sometimes impossible. Our research leverages a vast dataset provided by Alibaba [1] to identify performance-influencing features, thereby streamlining logging and tracing strategies more effectively and reducing the volume

[1] https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022

of trace data needed for robust system analysis.

In this paper, we proposed a two-phase modeling process to identify and prioritize the most impactful features for CPU and memory utilization in a large-scale microservice platform, substantially reducing the volume of necessary trace data without compromising model accuracy. Our approach initiates by applying multiple rounds of preprocessing on the data to ensure quality. We then employ a Gradient boosting algorithm in a two-phase modeling approach: first, to identify key features affecting CPU and memory utilization, and second, to refine these features, aiming for a minimal yet effective set of features for performance modeling. This streamlined approach allows for significant reductions in trace data volume while maintaining the integrity of model performance. Our evaluation on the test data showed that while having about 69% reduction on the required data for analysis, we are able to achieve a RMSE score of 0.02 for modeling the CPU utilization and about 0.13 for modeling the memory utilization which are similar to the 0.08 and 0.14 achieved by using the full dataset, with slight improvement on the RMSE.

This paper makes the following contribution.

- To the best of our knowledge, our work is the first work to analyze the large-scale trace data in order to reduce the number of features required for efficient performance modeling.
- Our approach will significantly reduce the size of the data required for performance modeling without decreasing model efficiency.
- The outcome of our work can complement the use of current models to update the tracing decisions in microservice architecture.

While datasets at the scale of the Alibaba production cluster are rare in academic settings, they are increasingly common in large industrial environments operating hyperscale cloud or microservice infrastructures. Companies such as Alibaba, Google, Meta, and Microsoft routinely collect telemetry at this scale for operational monitoring and capacity planning. Our use of this dataset allowed us to test the approach under realistic, worst-case data volume conditions, demonstrating that the method is viable when trace data truly becomes a scalability bottleneck.

5.2 Related Works

We categorize the studies related to this work into two groups. The studies on the performance modeling approaches, and studies on assisting trace and log data reduction.

5.2.1 Assisting Trace and Log data reduction

Determining the optimal locations for logging within code is a critical challenge for developers, aiming to balance between over-logging, which can overwhelm system resources and generate a large amount of data, and under-logging, which risks missing crucial performance data [43]. Over the past years, studies have shown that developers tend to rely on their instincts when deciding tracing and logging locations in their code, which can be inefficient [95]. To correctly answering the "where to log" question, many studies have focused on different aspect of instrumentation over the pas few years, focusing on enhancing usability, diagnosability, and overall code quality.

Ding et al. present Log2 [22], a cost-aware logging framework that intelligently balances logging verbosity against a predefined budget. Similarly, Zhao et al. present Log20 [23], which seeks to optimize the informativeness of log statements by employing Shannon Entropy, taking into account the execution time and the number of generated log lines in specific period as a critical factor.

Further advancements in this field are seen in the development of Log4Perf [44], which specifically targets logging in web-based systems for performance analysis. This framework adopts a methodical approach by first identifying key performance-influencing factors through thorough testing and monitoring, followed by statistical analysis to pinpoint where logging should be implemented for maximum efficiency.

On the diagnosability field, various methodologies have been proposed to enhance failure diagnosis capabilities. Techniques range from static analysis, which scrutinizes the structure of the code for potential logging points [37], to dynamic analysis [109] that suggests logging locations based on the behavior of the system under simulated fault conditions. Notable efforts in this area include studies by Yuan et al. [37], which leverage static code analysis to improve failure diagnosis through the identification of critical exception patterns.

In addition, Locke et al. represent LogAssist [132], a novel approach to simplifying log analysis. By organizing logs into event sequences and employing n-gram modeling for data compression, LogAssist aims to reduce the log volume significantly, thereby streamlining the analysis process and reducing the time and effort required for developers to sift through log data.

These studies collectively contribute to a growing body of knowledge on effective logging practices, addressing the pressing need for balance in logging strategies to ensure comprehensive system monitoring without compromising performance or diagnosability. Nevertheless, most of the studies focus on analyzing the system in order to identify and suggest logging locations

which is not suitable or in some cases possible in microservice architecture. In contrast, our approach uses the existing trace data in order to identify the most important features for performance modeling, thus reducing the trace locations in the system.

5.2.2 Performance Modeling

The concept of using clustering from logging data to detect significant system states impacting performance, as introduced by Cohen et al. [40], which illustrates a foundational approach within this domain. This method, leveraging Tree-Augmented Bayesian Networks (TAN), underscores the potential for modeling system performance states based on a selected subset of metrics without extensive prior system knowledge. The exploration of Application Performance Management (APM) data for building performance models by Brebner et al [133] adds complexity and necessitates customization, reflecting the requirements for effective performance modeling in distributed systems.

Xiong et al. [134] propose an innovative solution for the automatic creation and selection of models based on diverse metrics, addressing the challenge of managing the large performance metrics inherent in microservice architectures.

Similarly, the methodology for efficiently grouping metrics into fewer clusters by Shang et al. [103] demonstrates advancements in scalable and adaptable performance benchmarking, enhancing model accuracy and efficiency through regression analysis. The use of statistical techniques like control charts to facilitate the analysis and communication of performance data signifies a broader effort within the academic community to refine tools and methodologies for microservice performance analysis.

The categorization of models into rule-based, data mining, and queuing frameworks by Gao et al. [135] expands the toolkit for addressing performance modeling challenges, offering a blueprint for future research in optimizing microservice architectures.

Recent studies significantly advanced our understanding and management of these complex systems. Bao et al. [136] introduce a method for performance modeling and workflow scheduling that aims to minimize end-to-end delays within budget constraints, highlighting the importance of efficient scheduling in microservices. Highlighted by Heinrich et al. [137], the need for performance-aware testing, monitoring, and modeling specific to microservices underlines the unique challenges and research directions necessary for these distributed systems.

Moreover, Jindal et al. [138] present Terminu, a tool for microservice capacity estimation using regression models, emphasizing the utility of the tool in achieving precise capacity

planning and performance optimization.

The rich usage of performance modeling supports our approach that leverages such model to identify the most important features for future tracing in microservice architecture.

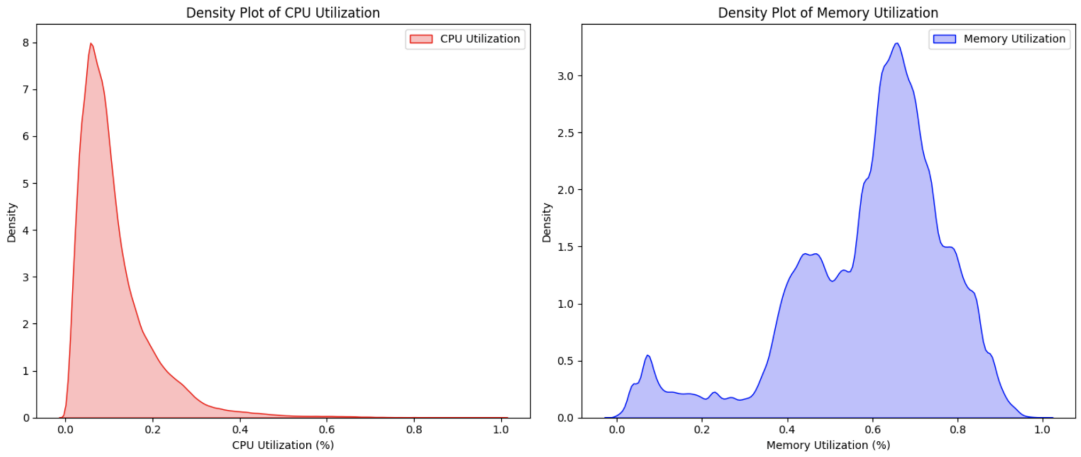

Figure 5.1 Overview of CPU and memory Utilization for a sampled period.

5.3 Methodology

In this section, we outline the detail of our methodology in reducing trace data required for modeling the CPU and memory utilization leveraging machine learning within a large-scale microservice architecture, similar to that used by Alibaba. In our study we used a comprehensive dataset obtained from the production cluster of Alibaba, that was made available for public[2]. This dataset spans over ten thousand bare-metal nodes, gathered across a span of 13 days and available for download in hourly intervals.

This dataset, which provides a real-world workload of the Alibaba production cluster is consist of four distinct parts. These include runtime data for over 40,000 bare-metal nodes, capturing CPU and memory utilization; runtime data for more than 470,000 containers across 28,000+ microservices, detailing their CPU and memory usage that are normalized using min max method between 0 and 1; microservice call rate and response time data, documenting interactions among 28,000+ microservices within the same ecosystem; and detailed call graph information, highlighting the communication patterns among 17,000+ microservices across multiple clusters [139]. Figure 5.1 illustrates the CPU and memory utilization of a sampled period of data. From these figures, we can observe the distribution of memory and CPU

[2]https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022

usage between 0 and 1, which gives us a good understanding of the resource usage in the provided dataset.

To pinpoint critical features for modeling CPU and memory utilization within microservices, we employed a two-stage process utilizing the Gradient boosting algorithm. The overall workflow of our methodology is illustrated in Figure 5.2. Initially, we trained a preliminary model using a subset of the dataset to identify statistically important features influencing CPU and memory utilization. Subsequently, we used a different dataset segment, previously unused, to trained a second model focusing solely on these identified features. This iterative method aimed to refine the feature set to the most statistically impactful ones, ensuring efficient performance modeling while minimizing the trace data required. The specifics of the process are outlined below, detailing our implementation approach.

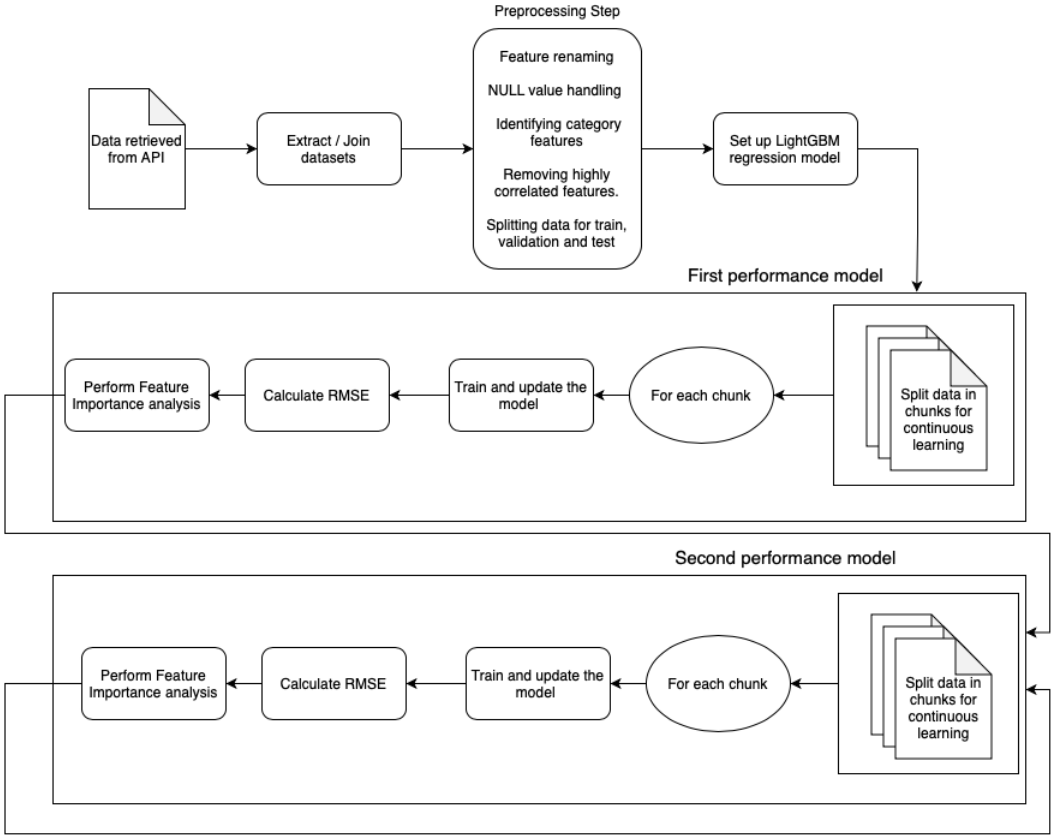

Figure 5.2 Overall workflow of our two step performance model.

5.3.1 Data Preprocessing

For the data preprocessing phase, we initially selected two hours from the first day to train our primary model and two hours from the second day for the secondary training data of

the model. The reason behind this selection is to reduce the data to a manageable sizes, which is still more than 100 gigabytes for a two hour window. Upon acquiring the necessary dataset portions, we undertook several preprocessing steps to enhance the data quality for our performance modeling efforts.

The initial step involved joining the aforementioned four datasets and creating a unified dataset for the performance modeling algorithm. We began this process by merging the MSResource dataset, which details microservice runtime information, with the MSRTMCR dataset — encompassing microservice call rates and response times and the MSCallGraph dataset, which illustrates the call graph interactions among microservices. The merging was based on the *msname* as a common identifier across these tables. Subsequently, we joined the Node dataset using *nodeid* as the joining key with the other three dataset. This comprehensive dataset merge facilitates a more comprehensive approach to performance modeling by leveraging the combined insights from all dataset components.

After performing the join on the four datasets, we first renamed the features in order to remove any character that might not be compatible with our model or could cause issues in analysis. This is particularly important for ensuring consistency in column names and avoiding errors related to invalid characters. Then, we identified the categorical features and converted them to the *category* data type. This conversion is essential for the model to correctly handle categorical features without needing one-hot encoding, allowing for more efficient use of memory and potentially improving model performance. The next step in data preprocessing is handling missing values. In this step, we filled the missing values in the dataset using the median of each column. This approach is a common practice to handle missing data, ensuring that the model does not encounter any NaN values that could disrupt the training process.

Subsequently, we refined the dataset by removing certain features. This included target features intended for prediction and those exhibiting strong correlation, identified using the Pearson correlation coefficient [140] to pinpoint features with a correlation of 90% or higher. This step ensures the elimination of redundancy in the dataset, enhancing the accuracy and efficiency of the subsequent performance modeling.

5.3.2 First Performance Model

For the first round of modeling the CPU and memory utilization of our dataset, we took advantage of the LightGBM algorithm [141]. LightGBM (Light Gradient Boosting Machine) is an efficient and scalable implementation of the gradient boosting framework. It is designed to be distributed and efficient with faster training speed and higher efficiency, lower memory

usage, and better accuracy. LightGBM extends the gradient boosting algorithm by introducing two novel techniques: Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB), which collectively improve on both the efficiency and effectiveness of the model.

Compared to other gradient boosting frameworks, such as XGBoost [142] or traditional Gradient Boosting Machines (GBM), LightGBM stands out for its efficiency and speed, especially on large datasets. It achieves a competitive accuracy with significantly less computational resources and time. Unlike XGBoost, which grows trees depth-wise (level-wise), LightGBM grows trees leaf-wise, selecting the leaf with the highest loss decrease for growth, which often results in better performance. One of the notable advantages of LightGBM is its native support for categorical features, eliminating the need for extensive preprocessing like one-hot encoding. LightGBM can handle categorical features by internally converting them into numerical values, considering the intrinsic order of categories when splitting nodes, which can lead to more efficient and accurate models, especially when dealing with high cardinality categorical data.

To model CPU and memory utilization, we configured the LightGBM model with specific parameters. The objective was set to *Regression* for accurate prediction, with a *learning rate* of 0.1 to ensure better generalization across data. We chose a *num_boost_round* of 5,000 for comprehensive learning. We then set the *max_depth* to 7 in order to prevent complex trees, while keeping the *num_leaves* at default values to balance model complexity and performance. To further avoid overfitting, we set the *labmda_l2* regularization to be 0.1. This setup aims to optimize the accuracy of the model in predicting resource utilization without overfitting to the training data.

Given the extensive data volume from merging four datasets, it was not practical to load all data into memory for training. Therefore, we employed pandas library capability to process the data in manageable chunks. This approach involved training the model incrementally, with each data chunk being split into three segments: 15% for testing, and of the remaining 85%, 90% was used for training while the final 10% served for validation. In each chunk of data, we used the bagging (Bootstrap Aggregating) capabilities of LightGBM in order to reduce variance and avoid overfitting. The validation set from each chunk was crucial for the training function, incorporating an early stopping mechanism set at 50 rounds to halt training if no improvement was observed, optimizing both memory usage and model performance.

Post-training and validation on each data chunk, we evaluated the model with test data, focusing on two aspects: calculating the root mean square error (RMSE) to assess model

performance, and ranking features by their importance, using the *gain* importance type from LightGBM as a criterion. This method highlighted the most critical features within each chunk. Subsequently, we updated the model, the list of feature importance, and the mean square error accordingly, ensuring continuous refinement and accuracy enhancement of the model predictive capabilities.

5.3.3 Second Performance Model

After applying our methodology to an initial dataset, we proceeded to extend this approach to a second dataset, adhering to the same parameters and data distribution principles established with the initial LightGBM model. In this subsequent phase, we began with the significant features identified from the first dataset, methodically reducing the number of features to retain only those with the highest statistical impact. This process enabled us to evaluate the contribution of each individual feature towards the capability of the model in accurately predicting CPU and memory usage.

Continuing with our iterative refinement, we focused on identifying the most streamlined yet effective combination of features. We conducted experiments with different subsets of top-ranked features to identify the best group of features, including, but not limited to groups of the top 9, top 5, and the top 3 features, each one chosen based on their demonstrated importance from the previous analysis. Our objective throughout was to discover the minimal set of features that could reliably predict system performance without a noticeable drop in accuracy. This step was crucial to optimize the efficiency of the model and ensure its practical applicability, and we present the results for the top 9, top 5 and top 3 features as representative examples.

Through this process, we were able to distill a set of features that, when used, delivered performance modeling results comparable to using the entire dataset. This discovery is crucial for future tracing efforts, as it suggests that similar model performance can be achieved with a significantly reduced dataset, focusing only on the most impactful features.

5.4 Result and Analysis

In this Section, we present the result of our methodology along with the detailed analysis of the results.

The results of our analysis are detailed in table 5.1. We assessed the performance of our model using the Root Mean Square Error (RMSE), employing a holdout method for evaluation. Specifically, about 15% of the data from each training chunk was reserved for testing the

Table 5.1 Result of our two phase performance model.

Number of features	RMSE CPU	RMSE Memory	Data reduction(%)
All 29	0.08	0.14	0 (full data)
Top 9	0.02	0.13	69 %
Top 5	0.14	0.21	83 %
Top 3	0.28	0.35	90 %

model. This approach allowed us to thoroughly evaluate the accuracy and effectiveness of the model in predicting outcomes based on the training data provided.

The analysis table shows that with the complete dataset, our model achieved an RMSE of 0.08 for predicting the CPU utilization, indicating that on average, the predicted values are 0.08 units away from the actual value, which can be acceptable in the field of CPU usage. Additionally, for memory utilization, an RMSE of 0.14 was noted, suggesting that on average, the predictions of the model were 0.14 units away from the actual values in the test dataset.

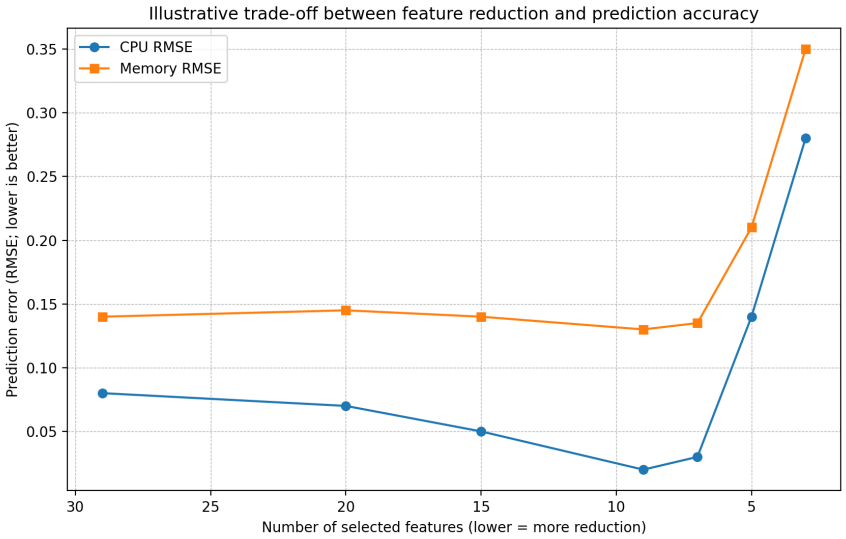

Figure 5.3 The trade-off between data reduction and prediction accuracy.

When we refined our analysis to only include the top 9 most important features, we noticed an improvement in prediction accuracy, with the RMSE for CPU utilization dropping to 0.02, marking a 0.06 increase in accuracy, and for memory utilization, the RMSE improved to 0.13, a 0.01 accuracy enhancement. This reduction of features from 29 to 9 led to a substantial decrease in the dataset size by approximately 69%, significantly affecting various aspects of the analysis like the memory needed, processing time, and energy usage.

Further refining our approach by focusing on the top 5 features, identified from the importance rankings of the previous step, led to a significant 83% reduction in data size. This strategic narrowing only minimally impacted model accuracy, with the RMSE for CPU utilization recorded at 0.14 and for memory utilization at 0.21. Such levels of accuracy, given the substantial decrease in data volume, may be deemed acceptable in various scenarios, demonstrating the effectiveness of targeted feature reduction in maintaining model performance with drastically less data.

When we narrowed down the feature list to the top 3 features based on their significance further reduced data size by approximately 90%, significantly lowering resource needs for modeling microservice performance. However, this substantial reduction led to increased RMSE values for both CPU and memory utilization, recording at 0.28 and 0.35, respectively. This indicates a notable decrease in prediction accuracy, highlighting the trade-offs between minimizing data volume and maintaining model precision. As mentioned, the top 9, 5 and three features were selected for demonstration purposes. The overall trend of the trade-off between the data reduction and prediction accuracy in our analysis can be seen in Figure 5.3.

Table 5.2 presents the top 16 features identified for their significance through the feature importance analysis of the LightGBM model. The features selected as most impactful in CPU and memory utilization models, are selected from this list. Notably, feature names appended with *_rt* and *_mcr* indicate their association with return time and call rate metrics, respectively. Return time is measured in milliseconds, while call rates are normalized on a scale from 0 to 1, employing min-max normalization for consistency across different communication paradigms.

From the analysis presented in table 5.2, it is evident that the most impactful features influencing CPU and memory utilization are closely tied to communication, memory, and database interactions within microservices. The sorted top 9 features for CPU performance modeling are http_rt, providerrpc_rt, consumerrpc_rt, writedb_rt, readdb_rt, consumermq_rt, providermq_rt, http_mcr, and providerrpc_mcr. The sorted top 9 features contributing to memory performance modeling are readmc_rt, writemc_rt, readmc_mcr, readdb_rt, writedb_rt, consumermq_rt, providermq_rt, rpc_id, and dminstanceid, which are selected from Table 5.2.

This observation aligns with the fundamental principles of microservice architectures, where the functionality of a system heavily relies on the continuous exchange of information between nodes. Such findings further validate our methodology, emphasizing that a focus on data related to communication, memory, and database access can noticeably decrease the volume

Table 5.2 Top 16 features used in modeling CPU and memory utilization

Feature	Description
consumermq_rt	Return time of fetching message from queue
dminstanceid	Container ID of an upstream microservice(MS)
writedb_rt	Return time of writing on database
readdb_rt	Return time of reading from database
interface	Interface of call from upstream to downstream MS
http_mcr	Rate of HTTP calls
http_rt	Return time of HTTP calls
readmc_rt	Return time of reading Memcached
providermq_rt	Return time of writing message to queue
providerrpc_rt	Return time of RPC calls for provider
consumerrpc_rt	Return time of RPC calls for consumer
readmc_mcr	Rate of reading Memcached
rpc_id	Unique ID of RPC call
providerrpc_mcr	Rate of RPC calls for provider
writemc_rt	Return time of writing to Memcached
readdb_mcr	Rate of reading from Database

of data needed for accurate modeling of CPU and memory utilization in these environments. By focusing in on these critical areas, our approach not only streamlines the data collection process but also enhances the precision of our performance models, stating the efficiency of targeted data analysis in complex distributed systems.

Concentrating on collecting data from the most critical features have multiple advantages. Firstly, it significantly reduces the need to insert numerous trace points within the system, thereby substantially lowering the data collection overhead of the system. This reduction in trace points directly leads to decreased CPU usage, memory consumption, and hard disk space required for storing the accumulated data [85]. Secondly, the process of analyzing a more focused dataset necessitates fewer computational resources. This streamlined analysis benefits from reduced memory requirements, quicker processing times, and lower energy consumption, as it avoids the extensive computational demand typically associated with processing large volumes of trace data. By prioritizing efficiency in both data collection and analysis, our approach enhances the overall resource management, leading to a more sustainable and cost-effective operation.

5.5 Discussion

In this study, we explored a two-step method to analyze trace data from large-scale microservice systems like Alibaba, aiming to minimize data needed for precise performance analysis without sacrificing accuracy in modeling the CPU and memory utilization. While our methodology is potentially generalizable to various large-scale systems, its effectiveness is inherently system-dependent, necessitating special implementation for each new environment. In our methodology, we employed the LightGBM algorithm, which has many advantages when handling complex datasets with both categorical and numerical features. Nevertheless, the correctness of our results relies on the accuracy of the trained model. To counter potential overfitting, we adjusted parameters required to train the LightGBM model, aiming for a balance between accuracy and efficiency.

The reason behind using LightGBM model solely in our evaluation is due to its exceptional capability to perform both predictive modeling and feature importance analysis, crucial for our objective of efficiently managing large-scale datasets with extensive features, in addition to identifying the most statistically impactful features in the predictive model. Unlike other algorithms such as Principal Component Analysis (PCA) [143], Recursive Feature Elimination (RFE) [144], and Genetic Algorithms (GA) [145], LightGBM stands out for its efficiency and effectiveness in handling high-dimensional data and providing insightful feature importance metrics. This approach ensures our model's accuracy and interpretability, focusing on the strengths of LightGBM and the proven performance in similar contexts, which inherently addresses the challenges associated with large data volumes and numerous features.

While we have shown the capability of our approach in reducing data requirements for the Alibaba dataset, testing the approach on other real-world applications with the goal of validation in both operational settings and evaluation on different datasets are essential to fully assess the capability of our method in understanding resource use, underscoring the need for further practical testing.

This study offers novel insights for both the research community and system engineers, providing a foundation for further large-scale trace analysis and performance modeling approaches. Researchers are encouraged to build upon these findings to enhance trace analysis methodologies, while engineers can adopt this approach to improve data collection and analysis within their microservice systems. This dual applicability promotes efficiency and accuracy in both academic and practical applications, fostering innovation in performance modeling and system optimization.

In concluding our discussion, it is important to mention that comparing the effectiveness of

our approach, which utilizes a two-phase analysis of trace data to reduce trace volume, with studies that applied a similar methods is challenging. This is primarily because, to the best our knowledge, our study is the first to answer the question of which part of the system to trace by analyzing the existing trace data and identifying the most impactful features. As such, there are no existing studies that closely align with ours for a direct comparison. This situation highlights the innovative aspect of our work and establishes a starting point for future research in this area.

5.6 Conclusion and Future Work

In our research, we proposed a two-phased approach to performance modeling that leverages gradient boosting techniques, specifically utilizing the LightGBM model. This strategy aims to streamline the feature set necessary for accurate modeling of CPU and memory utilization within large-scale microservice environments. In our study, we used the trace data collected from Alibaba production cluster. Initially, our methodology employs the LightGBM model to precisely estimate CPU and memory usage, achieving root mean square error (RMSE) values of 0.08 and 0.14, respectively. Subsequently, we refine our approach by reducing the dataset size by approximately 69%, concentrating on the most statistically influential features. This refinement not only simplifies the model but also enhances the accuracy, illustrated by improved RMSE scores of 0.02 for CPU and 0.13 for memory utilization.

Our findings illustrate that by prioritizing critical components of microservice architecture—specifically, inter-service communication, along with memory and database interactions—it is possible to effectively model and predict resource usage in large-scale microservice networks with significantly reduced data. This methodological advancement underscores the potential for more efficient and accurate performance modeling, enabling better resource management in complex microservice ecosystems.

Looking forward, we aim to further explore the scalability and applicability of our approach across different microservice environments and configurations. Future work will focus on validating the robustness of the model in varying operational conditions, potentially integrating real-time data streaming to enhance predictive capabilities. Additionally, investigating the integration of our methodology with automated system monitoring tools represents a promising direction, aiming to provide system administrators and developers with more agile and responsive tools for performance management.

Acknowledgements

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Ciena, Ericsson, and EffciOS for funding this research project.

CHAPTER 6 ARTICLE 3: LOGGING PRACTICES IN SOFTWARE BUG RESOLUTION: AN EMPIRICAL STUDY

Authors: Amir Haghshenas, Naser Ezzati-Jivan, Michel Dagenais

Venue: Empirical Software Engineering

Submission Date: 17-10-2025

Abstract: Logging is widely used in software development for monitoring system behavior and diagnosing issues. While previous research has examined logging practices and their characteristics, there is limited understanding of logging patterns in bug-fixing activities over time. This paper investigates the relationship between logging practices and bug resolution through a mixed-methods empirical study. We analyzed the commit histories of ten popular open-source Java projects on GitHub, examining over 572,000 functions across projects including Apache Hadoop, Flink, Jenkins, Kafka, Cassandra, Maven, Elasticsearch, Tomcat, Spring Framework, and Eclipse Trace Compass. Our analysis categorizes commits into bug-fixing and non-bug-fixing groups to understand how logging patterns differ between these contexts. Additionally, we conducted a survey of 58 developers from both industry and open-source communities to understand their perspectives on logging decisions and practices. Our findings reveal significant variation in logging patterns across projects, with logging coverage ranging from 3.9% in CI/CD platforms like Jenkins to 25.5% in infrastructure projects such as Hadoop. Bug-fixing functions show higher logging coverage in most projects, with developers adding logging 1.2 to 3.8 times more frequently during bug fixes. The association between logged functions and bug involvement varies across projects: while infrastructure projects show strong positive associations (odds ratios of 1.21-1.48), framework and utility projects show weaker associations (odds ratios near 1.0). The survey results indicate a gap between developer intentions and actual practices: while 74.1% of industry developers claim to prefer a balanced approach to logging, our code analysis shows predominantly reactive logging patterns. Industry developers prioritize log clarity and business context, while open-source developers focus more on measurable outcomes like bug resolution time. These findings present empirical evidence about logging patterns in bug resolution contexts, showing associations between project domain and complexity and observed logging patterns.

6.1 Introduction

Software development increasingly relies on logging practices to monitor system behavior, diagnose issues, and maintain software quality in complex, distributed environments [37, 146]. Logging involves systematically recording events, errors, and state information during program execution, creating diagnostic trails that developers use for debugging, performance monitoring, and system understanding [147]. As software systems grow in complexity and deployment scale, logging has become widely adopted for post-deployment maintenance and reliability [27, 148].

Within software maintenance, the relationship between logging practices and bug resolution represents a critical but poorly understood area. Early empirical research established foundational understanding of logging practices through large-scale studies of open-source and industrial systems [28, 146]. These studies revealed that logging decisions heavily depend on developer domain knowledge and lack rigorous specifications, with significant variations across different software contexts. Subsequent research has characterized logging practices across different domains, showing that only 25% of exception handling code and 9% of return-value checks include logging statements [95]. These findings highlight systematic gaps in current logging practices and suggest opportunities for improvement. Recent advances have focused on automated approaches to support developer decision-making in logging. Machine learning techniques have been developed to suggest appropriate logging locations within code blocks [149], predict suitable log levels using neural networks [150], and recommend which variables to include in log statements [151]. Studies using topic modeling have demonstrated that actual software functionality drives logging decisions, with network communication methods being more likely to be logged than simple operations [148]. However, these automated approaches primarily learn from existing practices without validating whether current logging strategies effectively support bug detection and resolution.

Research has also revealed project-specific variations in logging practices. Studies of mobile applications show different logging characteristics compared to server-side systems [152], while analysis of Apache Foundation projects demonstrates significant variation across different software categories [27]. Industrial studies indicate that enterprise systems have different logging requirements and practices compared to open-source projects [146]. These variations suggest that logging patterns may be context-dependent, but the relationship between logging practices and actual debugging results remains unclear.

Despite the assumed benefits of logging for debugging, there is limited empirical evidence on how logging practices relate to bug resolution outcomes. Existing research presents counterintuitive findings: files with logging statements sometimes exhibit higher defect densities, and bug reports containing log messages may take longer to resolve [27, 153]. These contradictory results suggest that the relationship between logging and debugging effectiveness is more complex than commonly assumed. Critically, we lack understanding of whether functions with logging are more prone to bugs, how developers modify logging during bug fixes, and whether specific logging patterns correlate with successful debugging outcomes. This knowledge gap prevents practitioners from making evidence-based decisions about when and how to implement logging strategies.

This study provides the first large-scale empirical analysis of the relationship between logging practices and bug resolution through mixed-methods investigation. We analyzed over 572,000 functions across ten popular open-source Java projects, examining how logging patterns differ between bug-fixing and non-bug-fixing commits. Additionally, we surveyed 58 developers from industry and open-source communities to understand their logging decision-making processes. Our research addresses three key questions:

RQ1: What is the relationship between bug-fixing commits and logging in terms of quantity, levels, and modifications?

RQ2: What proportion of files with logging statements are involved in bug-fixing commits compared to those without logging?

RQ3: How do developers perceive and approach logging decisions in bug-fixing contexts, and how do these perceptions align with actual logging practices?

Our findings reveal significant variation in logging associations across project domains, with infrastructure projects showing positive associations between logging and bug involvement (odds ratios 1.21-1.48) while framework and utility projects show weaker associations (odds ratios near 1.0). We also identify a substantial gap between developer intentions and actual practices, with 74.1% of industry developers claiming balanced logging approaches while code analysis reveals predominantly reactive patterns.

The contributions of this work are threefold. First, we present empirical evidence about logging patterns in bug-fixing contexts through large-scale analysis of Java open-source projects,

revealing significant variation across project domains and types that establishes baseline patterns for the Java ecosystem. Second, we document the gap between developer intentions and actual logging practices by comparing survey responses with observed behaviors in GitHub repositories, providing insights into Java development practices. Third, we identify associations between logging patterns and bug resolution activities in Java projects, showing how these associations vary across project-specific factors and establishing patterns that warrant investigation in other programming contexts.

The remainder of this paper is organized as follows: Section 6.2 reviews the related work on logging practices and their impact on software maintenance. Section 6.3 presents the motivation examples of the study. Section 6.4 describes our mixed-methods methodology for data collection and analysis. Section 6.5 presents the results of our empirical analysis and developer survey. Section 6.6 discusses the implications of our findings and provides recommendations for practitioners. Section 6.7 covers the threats to validity of our work, Finally, Section 6.8 concludes the paper and suggests directions for future research.

6.2 Related Works

The evolution of software logging research spans over a decade, progressing from foundational empirical studies to sophisticated automated approaches. This section traces the chronological development of logging research, highlighting key findings and technological advancements.

Systematic logging research began with Yuan et al. [154], who introduced log enhancement for improved software diagnosability. Their LogEnhancer tool demonstrated how automated enhancement could reduce potential failure causes during diagnosis. Yuan et al. [28] conducted the first large-scale empirical study, analyzing four C/C++ server-side projects and establishing foundational findings about logging practices, including the observation that developers often add logging reactively after failures.

Fu et al. [146] transitioned research to industrial contexts, analyzing two large Microsoft systems and surveying 54 developers. They achieved 90% accuracy in predicting logging locations but critically found that logging decisions lack rigorous specifications and depend heavily on developer domain knowledge. Zhu et al. [95] introduced the first automated logging assistance with LogAdvisor, revealing that only 25.3% of exception snippets and 9.3%

of return-value checks were logged, highlighting systematic gaps in logging practices..

Chen and Jiang [27] conducted a replication study of 21 Java-based Apache projects, confirming significant variation across project categories. Importantly, they found that bug reports with log messages take longer to resolve than those without—challenging assumptions about logging's debugging benefits. Li et al. [148] employed topic modeling to demonstrate that software functionality drives logging decisions, with network communication methods being more likely to be logged than simple operations.

This period also revealed counterintuitive relationships between logging and software quality. Shang et al. [153] found that files with logging statements had higher post-release defect densities in 7 out of 8 studied releases, questioning conventional assumptions about logging's relationship to code quality.

Research shifted toward developing automated tools to support logging decisions. Zhao et al. [23] developed Log20, optimizing logging placement using information theory. Gholamian [155] leveraged NLP and code clones for automated log placement prediction. Li et al. [156] introduced DLFinder to detect duplicate logging code smells, while specialized tools like Log4Perf [94] emerged for performance monitoring. Li et al. [1] also addressed log quality, proposing guidelines to improve message clarity and structure. Recent work has integrated sophisticated machine learning approaches. Li et al. [150] advanced automated logging with "DeepLV," using neural networks for log level selection with cross-system suggestions outperforming within-system approaches. Li et al. [149] provided block-level guidance, identifying six logging location categories and developing deep learning frameworks for statement-level suggestions.

Cutting-edge developments include Ding et al. [157], who introduced LoGenText for automatic log message generation using neural machine translation, and recent systems [158] applying LLMs for comprehensive logging automation with in-context learning. Recent research explored specialized contexts. Zeng et al. [152] studied mobile applications, finding that mobile logging is less pervasive than server applications with different characteristics. Zhang et al. [159] examined test code logging, showing that test logs serve different purposes than production logs.

Developer perspective studies include Li et al. [1], who interviewed developers about logging

trade-offs between debugging value and performance overhead, and Li et al. [160], who identified three critical aspects of log message readability through practitioner interviews. Batoun et al. [161] systematically reviewed 204 papers and 20,766 StackOverflow questions, finding that 53% of studies focus on log preprocessing while 37% focus on instrumentation.

Despite significant advances, critical gaps remain. While numerous studies have characterized logging practices and developed automated tools, there is limited empirical evidence on how logging practices actually influence bug resolution outcomes. The counterintuitive findings about logged files having higher defect densities [153], and bug reports with log messages taking longer to resolve [27], suggest a more complex relationship than commonly assumed.

Existing research has focused on describing current practices or learning from them without validating their effectiveness for bug detection and resolution. The field lacks understanding of whether functions with logging are more prone to bugs, how developers modify logging during bug fixes, and whether specific patterns correlate with successful debugging. While studies have noted project-specific variations [27, 152], there has been no systematic investigation of how logging effectiveness varies across software domains.

This study addresses these gaps by providing the first large-scale empirical analysis specifically focused on the relationship between logging practices and bug resolution. By analyzing over 308,000 functions across diverse projects and combining this with developer surveys, we offer evidence-based insights into when and how logging contributes to effective bug resolution.

6.3 Motivation

The substantial investment of the software industry in the debugging and logging infrastructure reflects the critical importance of these activities for software reliability. A comprehensive study by Cambridge Judge Business School found that debugging costs the global software industry $312 billion annually, and software developers spend approximately 50% of their programming time finding and fixing bugs [1]. This massive economic burden has driven explosive growth in logging infrastructure markets, with the global log management market projected to grow from $2.3 billion in 2021 to $7.0 billion by 2030, representing a compound

[1] https://www.jbs.cam.ac.uk/2013/research-by-cambridge-mbas-for-tech-firm-undo-finds-software-bugs-cost-the-industry-316-billion-a-year/

annual growth rate of 10.4% [2]. These figures demonstrate the recognition in industry that effective logging and debugging capabilities are essential for maintaining software quality and operational reliability.

Academic research has begun to systematically examine the effectiveness of these logging investments, revealing fundamental limitations in current approaches. A comprehensive survey of automated log analysis research found that logs often contain insufficient information for automatic detection of failures or root cause diagnosis, with small events potentially generating dramatic log volume variations while critical failures may produce minimal logging signatures [162]. Large-scale industry analysis confirms these findings, with studies of 2.3 million log entries from critical industrial systems showing that event logging lacks systematic design and implementation practices, with logging effectiveness strongly dependent on human expertise rather than systematic engineering approaches [109]j.

Despite these massive investments in logging infrastructure, critical software failures continue to occur with devastating consequences, often evading detection by sophisticated monitoring systems. The 2021 Facebook global outage illustrates how logging can detect symptoms without identifying root causes: Facebook, Instagram, and WhatsApp were down for 6+ hours due to a BGP routing configuration error, but extensive network monitoring systems detected route withdrawals without determining the underlying cause, with internal monitoring systems becoming unreachable when the network was isolated [3]. Similarly, the 2022 Atlassian mass site deletion incident saw 775 customers lose access to their products for up to 14 days when a script intended to delete legacy apps instead deleted entire customer sites, yet the comprehensive application monitoring of the company failed to detect the issue because their internal monitoring did not detect an issue because the sites were deleted via a standard workflow. [4]

These cases reveal a fundamental disconnect between logging investments and debugging effectiveness: sophisticated monitoring systems can miss catastrophic errors when they occur through normal operational channels or when circular dependencies compromise the monitoring infrastructure itself. This disconnect suggests that current logging practices may not be optimally designed for bug detection and resolution, despite their widespread adoption and the substantial resources devoted to their implementation. The combination of massive in-

[2]https://www.businessresearchinsights.com/market-reports/log-management-software-market-111895
[3]https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/
[4]https://www.atlassian.com/blog/atlassian-engineering/post-incident-review-april-2022-outage

dustry investment, growing market demands, and continued critical failures in well-monitored systems indicates a need for empirical research into the actual relationship between logging practices and bug resolution effectiveness. This evidence motivated our investigation into the relationship between logging practices and bug resolution outcomes. While the industry continues to invest billions in logging infrastructure based on the assumption that comprehensive logging improves debugging effectiveness, there is limited empirical evidence validating this assumption. Our study addresses this gap by analyzing over 308,000 functions across nine popular open-source projects to understand how logging patterns actually relate to bug-fixing activities, providing evidence-based insights into when and how logging contributes to effective software maintenance.

6.4 Methodology

Our study employs a mixed-methods approach to investigate the relationship between logging practices and bug resolution in software development. Our methodology combines large-scale quantitative analysis of source code repositories with qualitative insights from developer surveys. We designed the study to provide comprehensive empirical evidence about logging patterns across diverse software projects and development contexts.

6.4.1 Project Selection

We selected ten popular open-source Java projects from GitHub based on specific criteria that ensure comprehensive coverage of different software domains and development patterns. We chose Java as the target language due to its widespread adoption in enterprise and open-source development, mature ecosystem, and consistent logging frameworks across projects. Our selection criteria included: (1) long-term development timeline spanning multiple years to capture evolution in logging practices, (2) high contribution rates indicating active development communities, (3) diverse software domains to examine logging effectiveness across different contexts, (4) substantial codebase size to provide statistically significant analysis samples, and (5) projects primarily written in Java to ensure comparable analysis results.

Our selected projects represent a variety of software categories including enterprise application frameworks (Spring Framework), stream processing systems (Apache Flink), CI/CD automation platforms (Jenkins), big data processing frameworks (Apache Hadoop), distributed databases (Apache Cassandra), event streaming platforms (Apache Kafka), build and de-

pendency management tools (Apache Maven), search and analytics engines (Elasticsearch), web application servers (Apache Tomcat), and specialized monitoring tools (Eclipse Trace Compass). This diversity enables us to examine how project domain, complexity, and development patterns relate to logging practices and their associations with bug resolution across the Java ecosystem.

For each project, we cloned the complete repository to local storage and extracted the full commit history using standard Git commands in GitHub. This approach ensures access to complete development history including all branches, merge commits, and metadata necessary for comprehensive analysis. Table 6.1 presents the key characteristics of our selected projects based on publicly available repository statistics and our analysis.

Table 6.1 Selected projects characteristics

Project	Domain	Stars	Forks	Commits	Contributors
Spring Framework	Enterprise Framework	58,400	38,600	33,309	970
Apache Flink	Stream Processing	25,000	13,700	36,967	1,314
Jenkins	CI/CD Platform	22,200	9,100	37,093	805
Apache Hadoop	Big Data Framework	15,200	9,100	27,870	635
Apache Cassandra	NoSQL Database	9,300	3,700	31,187	464
Apache Kafka	Event Streaming	39,400	14,500	16,048	1,273
Apache Maven	Build Tool	4,700	2,600	15,659	212
Elasticsearch	Search Engine	73,200	25,300	87,818	2,047
Apache Tomcat	Web Server	7,900	5,200	27,443	132
Eclipse Trace Compass	Enterprise Framework	27	22	8,158	73

6.4.2 Commit Classification

We developed a classification system to distinguish bug-fixing commits from other development activities within our selected repositories. Our approach relies on multiple sources of information available in the commit history: commit messages and GitHub issue metadata. This multi-strategy classification ensures robust identification of bug-fixing activities while minimizing false positives from feature development or routine maintenance commits. Our classification process employs four distinct strategies that operate hierarchically. We designed this approach to capture the diverse ways developers communicate bug-fixing activities in

commit messages while accounting for variations in project conventions and individual developer practices.

Strategy 1: GitHub Issue Reference with Bug Labels. We cross-reference commit messages with GitHub issue metadata to identify commits that explicitly reference issues labeled as bugs. We extract issue references using regular expression patterns including "#123" (GitHub style), "fixes #789", and "issue 101". When a commit message references an issue number that appears in our dataset of bug-labeled issues, we classify it as bug-fixing.

Strategy 2: Revert and Emergency Fix Patterns. We identify commits that explicitly indicate urgent bug resolution through revert operations or emergency fixes. Our algorithm recognizes patterns including "revert", "rollback", "undo", "hotfix", "quickfix", and "emergency fix". These patterns typically indicate critical bugs that required immediate attention and represent high-confidence bug-fixing activities.

Strategy 3: Java-Specific Error Patterns. We developed a catalog of multi-word phrases commonly associated with Java programming errors and exceptions. Our list includes technical terms such as "null pointer", "index out of bounds", "stack overflow", "out of memory", "race condition", "connection timeout", "resource leak", "class not found", "concurrent modification", and "thread safety". These phrases directly indicate specific technical problems that developers encounter in Java applications.

Strategy 4: Action and Problem Keyword Combinations. We implemented a two-tier keyword matching system that requires the presence of both action keywords and problem keywords within the same commit message. Our action keywords include "fix", "resolve", "solve", "patch", "correct", "repair", "address", "handle", and "prevent" along with their various grammatical forms. Our problem keywords encompass "bug", "issue", "problem", "error", "crash", "failure", "exception", "fault", "defect", "broken", "deadlock", "regression", and "vulnerability". We require co-occurrence of at least one keyword from each category to classify a commit as bug-fixing.

Our classification algorithm processes each commit message by first converting it to lowercase and then applying the strategies in the hierarchical order described above. Strategy 1 receives highest priority due to its reliance on explicit issue labeling, followed by Strategies 2, 3 and 4 which indicate clear technical problems. Once a commit matches any strategy, we immediately classify it as bug-fixing without further analysis. This hierarchical approach ensures that the most reliable indicators take precedence while still capturing bug-fixing commits that may not follow standard conventions.

We implemented this classification system as a Python script that processes the complete commit history extracted from each repository. The script generates detailed statistics showing the distribution of commits across different classification strategies, enabling us to assess the effectiveness of our approach and identify potential areas for refinement. Our system also preserves the classification rationale for each commit, allowing for post-hoc analysis and validation of our results.

To validate our automated classification system, we conducted manual verification using stratified random sampling across all projects. We determined the required sample size using the standard sample size formula for population proportions: $n = (Z^2 \ddot{O} p \ddot{O}(1-p))/E^2$, where Z = 1.645 (Z-score for 90% confidence level from the standard normal distribution), p = 0.5 (maximum variance assumption when true proportion is unknown), and E = 0.10 (10% margin of error). This calculation yielded a minimum sample size of 68 commits per project, which we increased to 100 commits to ensure adequate representation across all classification strategies.

The first author manually assessed whether each sampled commit message expressed bug-fixing intent, focusing on the stated intention of the developer rather than the technical correctness of the implemented changes. This validation process evaluated whether our automated classification correctly identified the intent expressed in commit messages, GitHub issue references, and developer-communicated problem-solving activities. We could validate aspects such as accurate keyword detection (e.g., confirming that "fix null pointer exception" contains both action and problem keywords), correct issue reference extraction (e.g., verifying that "#123" was properly linked to a bug-labeled GitHub issue), and appropriate pattern matching (e.g., ensuring that "revert broken changes" was classified under emergency fix patterns). However, we could not validate whether the code changes actually resolved the reported problems, whether the identified "bugs" were genuine technical issues or user errors, or whether the implemented solutions were technically sound since these decisions require expert knowledge on the application. Our validation focused exclusively on the accuracy of text classification rather than the technical assessment of the code modifications themselves. Our automated classification achieved a mean accuracy of 91% in correctly identifying developer-expressed bug-fixing intent across all projects, confirming that our system reliably captures commits where developers explicitly communicated their intention to address bugs or system problems.

6.4.3 Commit Metadata Collection

Following our commit classification process, we extracted information about the files modified in each commit to enable function-level analysis of logging practices. This step identifies the source code files that underwent changes during bug-fixing and non-bug-fixing activities, providing the foundation for our analysis of logging patterns within individual functions.

We developed an automated extraction system that processes the classified commits from both categories to gather metadata about file modifications. For each commit, we collected the commit identifier, commit date, commit message, and the complete list of modified files. We implemented this extraction using Python scripts that interface with Git command-line tools through subprocess calls, executing the *git diff-tree –no-commit-id –name-only -r* command to retrieve the names of changed files without displaying content differences.

Our extraction process reads classified commits from CSV files and processes each commit individually to handle large repositories efficiently. We manage potential errors by assigning empty file lists to problematic commits rather than terminating the entire process. The extracted information is structured as JSON objects containing the commit identifier, date, message, and an array of changed file paths.

We generated separate datasets for bug-fixing commits and non-bug-fixing commits, facilitating comparative analysis of file modification patterns between the two categories. This separation allows us to examine whether certain types of files are more frequently modified during bug-fixing activities compared to general development work. The resulting datasets serve as input for our function-level analysis, where we examine the logging practices within the functions that were modified in each commit.

6.4.4 File Content Extraction

After identifying the changed files for each commit, we extracted the actual file contents to enable detailed analysis of code modifications and logging changes. This step captures the before and after versions of each modified Java file, along with the diff information showing the exact changes made during each commit.

We developed an automated extraction system that processes each commit and retrieves three versions of every modified Java file: the file content before the commit, the file content after the commit, and the diff showing the specific changes. We focused exclusively on Java files by filtering for files with ".java" extensions, ensuring our analysis remains within the scope of our selected programming language. For each commit, we created a structured di-

rectory hierarchy organized by commit ID and date, with subdirectories for each modified file.

We implemented this extraction using Git commands through Python subprocess calls. For file content retrieval, we used *git show commit_id:file_path* to extract files at specific commit states, capturing both the pre-commit version and post-commit version. For change analysis, we used *git diff parent_commit...commit_id – file_path* to generate diff files showing the exact modifications made to each file. This approach provides the detailed view needed to analyze how logging statements were added, removed, or modified during each commit.

Our extraction process handles potential errors such as files that were deleted, renamed, or inaccessible by continuing processing rather than terminating the entire operation. We organized the extracted files with descriptive prefixes ("before_", "after_", "diff_") to clearly distinguish between different versions of the same file. The resulting directory structure enables direct comparison between file states and facilitates automated analysis of logging patterns and code changes.

We processed both bug-fixing and non-bug-fixing commit datasets separately, creating distinct directory hierarchies for each category. This separation maintains the classification structure established in our previous steps and enables comparative analysis of how file modifications differ between bug-fixing and general development activities. The extracted file contents serve as the foundation for our function-level logging analysis, where we examine specific changes to logging statements within the context of broader code modifications.

6.4.5 Diff Parsing and Function Identification

To analyze logging changes at the function level, we developed a parsing system that identifies specific functions modified in each commit. This step bridges the gap between file-level changes and function-level logging analysis by extracting the exact functions that were altered during bug-fixing and non-bug-fixing activities.

Our parsing approach processes the three file versions (before, after, and diff) extracted in the previous step to identify modified functions. We implemented a multi-stage algorithm that first matches corresponding file sets (diff, before, and after versions of the same file), then parses the diff files to extract change locations, and finally identifies the functions containing those changes. This approach ensures we capture all function modifications while maintaining accurate mapping between changes and their containing functions.

We parse diff files using regular expressions to extract chunk headers and change information. Our parser identifies three types of modifications: additions (lines starting with '+'), deletions (lines starting with '-'), and modifications (combinations of additions and deletions within the same code block). For each change block, we extract the line numbers where changes occurred in both the before and after versions of the file, along with the context lines surrounding the changes. This detailed parsing enables precise identification of the code locations that were modified.

To identify the functions containing the changes, we employ a three-tier search strategy. First, we search within the changed lines themselves for Java function definitions using regular expression patterns that match Java method signatures including access modifiers, return types, method names, and parameter lists. If no function definition is found within the changed lines, we search the immediate context lines that precede the changes within the same diff chunk. Finally, if no function is identified in the immediate context, we search backward from the change location in the complete file until we encounter the nearest function definition.

Our function identification uses regular expression patterns specifically designed for Java method signatures. This pattern captures various combinations of access modifiers, return types, and method names while handling generic types and parameter lists. When a function definition is matched, we extract the method name using a focused pattern that captures the identifier immediately preceding the parameter parentheses.

We process additions and deletions separately to ensure accurate function identification in both pre-commit and post-commit versions of files. For additions and modifications, we identify functions in the after-commit version of files since these represent the final state of the code. For deletions, we identify functions in the before-commit version since these represent the original state before removal. This approach ensures we capture functions that were modified, regardless of whether they were added, deleted, or changed.

The output of this process is a list of function names paired with their corresponding file paths, separated by commit classification (bug-fixing or non-bug-fixing). This function-level identification enables our subsequent analysis to examine logging practices within specific functions that underwent changes, providing the granular detail necessary to investigate how logging patterns differ between bug resolution and general development activities. The identified functions serve as the target units for our logging analysis, where we examine how

logging statements are added, removed, or modified within the context of different types of development work.

6.4.6 Function-Level Feature Extraction and Logging Analysis

The final step in our methodology involves extracting detailed metrics from the identified functions to enable quantitative analysis of logging practices and code characteristics. This process transforms the source code of individual functions into structured data that captures both code complexity metrics and logging behavior, providing the foundation for our comparative analysis between bug-fixing and non-bug-fixing development activities.

We developed an automated feature extraction system that processes each identified function using Abstract Syntax Tree (AST) parsing through the srcML tool [5]. This tool converts Java source code into an XML representation, preserving the complete syntactic structure while enabling systematic analysis of code elements. This approach maintains the semantic relationships between code components, allowing us to analyze not just the presence of specific constructs but also their hierarchical relationships and context within functions. Our approach processes both the before-commit and after-commit versions of each function, capturing how logging practices and code complexity change during specific development activities.

Our feature extraction encompasses three primary categories of metrics that provide a comprehensive view of function characteristics. Code complexity indicators include lines of code (counted through semicolons and code blocks), number and nesting depth of loops (for, while, do-while constructs), function calls, try-catch blocks, and recursive function identification. We calculate nesting depth by identifying loops contained within other loops, providing insight into algorithmic complexity. Structural characteristics cover different statement types (expression, declaration, empty statements) and branching complexity through if-statements and switch-case constructs. The combination of these metrics creates a multi-dimensional complexity profile that enables us to control for function sophistication when analyzing logging patterns.

Logging behavior analysis identifies and categorizes logging statements across six standard levels: info, debug, trace, warn, error, and fatal. We implement a comprehensive detection system that recognizes multiple patterns commonly used in Java applications. Our detection

[5] https://www.srcml.org

logic searches for logging level indicators in function call parameters, method names, and argument lists, handling both explicit logging calls (e.g., logger.info()) and parameterized logging patterns (e.g., log(Level.INFO)). The system examines function call names for logging level keywords, analyzes parameter lists for logging level constants, and checks argument content for logging framework calls using string matching techniques that account for variations in method invocation syntax.

We process functions using srcML's command-line interface through Python subprocess calls, removing XML namespace declarations and parsing the resulting structure using ElementTree. The XML parsing approach allows us to maintain the complete structural information of the source code while enabling precise queries for specific elements. We traverse the AST using XPath-like queries to locate specific code elements such as function calls, control structures, and statement types. Our parsing strategy preserves the hierarchical relationships between code elements, enabling us to distinguish between different contexts where the same constructs might appear.

For logging detection specifically, we handle both case-sensitive and case-insensitive patterns to accommodate different project conventions and logging frameworks. We examine import statements to identify logging framework usage and adjust our detection patterns accordingly. The detection system captures not only the count of logging statements at each level but also preserves the complete text of each logging call, enabling qualitative analysis of logging message patterns and parameter usage. This dual approach of quantitative counting and qualitative content preservation supports both statistical analysis and detailed examination of logging practices.

The extraction process generates structured JSON output containing the function name, file path, commit information, and complete before/after metrics for each analyzed function. Each function entry includes complexity metrics, structural characteristics, and detailed logging information for both the pre-commit and post-commit states. This data structure enables direct comparison of how individual functions evolved during each commit, supporting our analysis of whether logging changes correlate with other code modifications and whether different patterns emerge between bug-fixing and non-bug-fixing activities.

This granular data structure supports our subsequent analysis of how logging practices correlate with code changes across different development contexts. The resulting dataset provides the quantitative foundation for our research questions, enabling statistical analysis of logging patterns and their relationship to bug resolution activities across our selected Java projects.

The comprehensive nature of the extracted metrics allows us to investigate not only whether logging practices differ between commit types, but also how these differences relate to function complexity and structural characteristics.

6.4.7 Data Collection Challenges and Error Handling

During our automated extraction process using GitHub APIs and Git commands, we encountered various data inconsistencies and errors inherent to large-scale repository analysis. These included file system inconsistencies where files referenced in commit metadata were not accessible at specified commit states, file path modifications due to renaming or relocating files during refactoring activities, and Git object accessibility issues where commands failed to retrieve specific file versions due to repository artifacts or incomplete operations.

Given the scale of our analysis spanning over 570,000 functions across ten large repositories, we implemented a pragmatic error handling approach. When encountering any of these errors, we systematically excluded the problematic commit, file, or function from our analysis rather than attempting manual resolution. This decision was based on the impracticality of manually resolving errors across hundreds of thousands of functions and the sufficiently large sample sizes that remained after exclusions.

The functions, files, and commits included in our final analysis represent those for which we could establish complete and reliable before-after comparisons using automated methods. While this approach necessarily excludes some potentially relevant data, it ensures the consistency and reproducibility of our analysis methodology while maintaining the statistical power necessary for our research questions.

6.4.8 Developer Survey Design and Implementation

To complement our quantitative analysis of logging practices in source code repositories, we conducted a comprehensive survey to understand developer perspectives and decision-making processes regarding logging in bug-fixing contexts. This survey component addresses our third research question (RQ3) by capturing the human factors and subjective experiences that influence logging decisions, enabling us to compare stated intentions with observed practices in our code analysis.

Survey Design and Structure

We designed a structured 17-questions questionnaire that investigated four primary dimensions: demographic background and experience, decision-making factors about logging, logging practices during bug resolution, and challenges with current logging approaches. The survey employs a mixed-question format that combines multiple-choice questions, 5-point Likert scales, and open-ended questions to capture both quantitative patterns and qualitative insights about developer logging practices.

Key constructs include: logging decision factors (function complexity, critical business use-cases, anticipated problem areas, reactive needs, integration points), proactive versus reactive logging preferences, logging association perceptions measured through Likert scales, contextual information practices regarding function parameters and state variables, post-bug-fix behaviors with logging statements, code structure interactions, organizational practices and team guidelines, and trade-off considerations between logging overhead and debugging value.

Target Population and Recruitment Strategy

We targeted software developers with active experience through two distinct recruitment channels to examine potential differences in logging practices between commercial and open-source development contexts while ensuring diverse representation across different organizational structures.

Industry Partner: Our industry recruitment was conducted in partnership with Ericsson, a multinational networking and telecommunications company headquartered in Stockholm, Sweden. Ericsson is a leading provider of information and communication technology infrastructure, employing over 100,000 people globally and developing large-scale, mission-critical software systems where logging and monitoring are essential for system reliability. We distributed survey invitations through organizational managers, who distributed the link internally within their respective teams. We received 27 responses from Ericsson developers across various roles and experience levels.

Open-Source Community: For open-source developer recruitment, we focused on active contributors to the same projects included in our quantitative code analysis. We randomly selected 600 contributors from the commit histories of our selected projects. This approach ensures that survey respondents have direct experience with the codebases we analyzed, en-

abling meaningful comparison between stated practices and observed behaviors. We received 31 responses, representing a response rate of 5.2%.

Survey Validation and Data Collection

Prior to full deployment, we conducted pilot testing with 5 software development practitioners to evaluate question clarity and completion time. We implemented the survey using Google Forms to collect responses, ensuring complete anonymity by not recording any personal information. Participants could withdraw from the survey at any time before submission, but once submitted, responses could not be removed due to the anonymous nature of data collection.

Our final dataset includes 58 responses (27 from industry, 31 from open-source), providing balanced representation across different development contexts. By combining this survey data with our large-scale code analysis of over 572,000 functions, we can identify meaningful patterns and discrepancies between developer intentions and actual logging practices. This mixed-methods approach enables us to understand both the quantitative patterns of logging behavior in real codebases and the qualitative factors that drive developer decision-making in logging practices during bug resolution activities.

6.5 Results

Our analysis of over 572,000 functions across ten major Java projects uncovers significant variation in logging associations across software domains. Logging coverage ranges from 3.9% in Jenkins to 25.5% in Apache Hadoop, with developers consistently adding logging more frequently during bug-fixing activities in most projects. This comprehensive empirical study provides large-scale evidence of domain-dependent logging patterns and their relationship to software debugging practices.

6.5.1 The Scale and Scope of Analysis

We conducted a comprehensive analysis across 572,172 functions spanning ten diverse Java projects, from large-scale systems like Elasticsearch (222,434 functions) and Apache Tomcat (94,310 functions) to specialized tools like Maven (10,142 functions) and Jenkins (12,778 functions). Our parallel developer survey captured perspectives from 58 practitioners representing both industry professionals (27 from Ericsson) and open-source contributors (31

from our analyzed projects). This dual approach enables direct comparison between stated logging practices and observed behaviors in real codebases, revealing significant patterns in logging associations across different software domains.

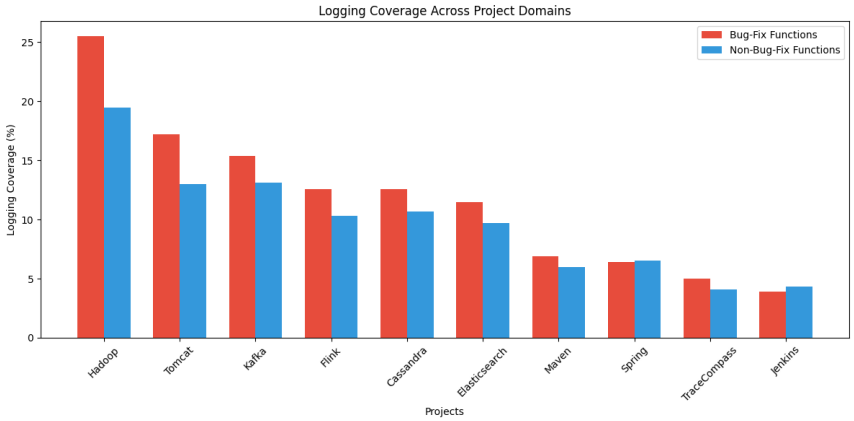

Figure 6.1 Logging coverage comparison between bug-fixing and non-bug-fixing functions .

6.5.2 RQ1: Logging Practices Across Software Domains

Logging Coverage

Figure 6.1 shows that the projects fall into three clear groups based on their logging coverage patterns. The first group includes Hadoop, Tomcat, and Kafka, which consistently show higher logging coverage in bug-fixing functions compared to regular development. Hadoop leads with a 6.0 percentage point difference, indicating that developers add significantly more logging when fixing bugs in distributed systems that handle large-scale data processing and complex failure scenarios.

The second group consists of Flink, Cassandra, Elasticsearch, and Maven, showing moderate but consistent differences between 0.9-2.3 percentage points. These projects handle complex operations but maintain more controlled environments compared to the first group. The third group includes Spring, Jenkins, and TraceCompass, where bug-fixing and regular development show similar logging coverage levels, suggesting that logging serves different purposes in these projects.

This clustering pattern shows associations between project complexity, operational environment, and logging patterns during debugging. Projects handling distributed operations show

higher frequencies of logging during bug fixes.

Logging Addition Strategy

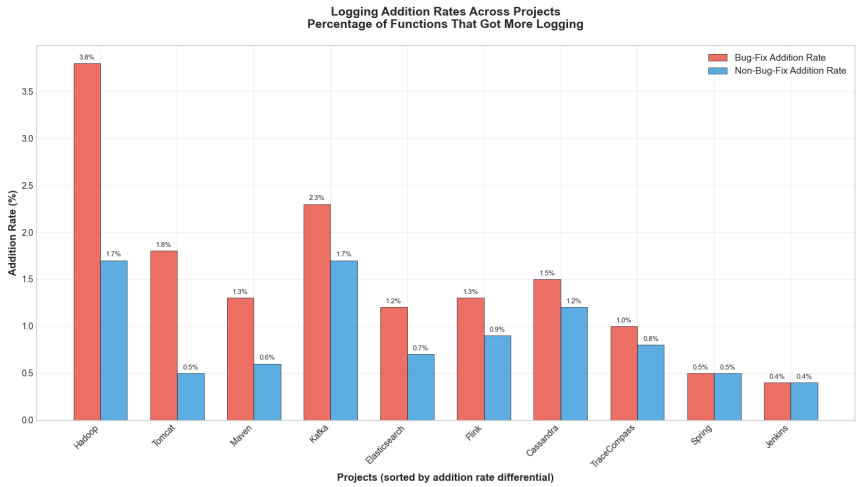

Figure 6.2 Increase in logging for both bug-fixing and regular development commits for each project

Figure 6.2 demonstrates that developers in different projects follow distinct strategies when adding logging during bug fixes. Hadoop and Tomcat show the largest gaps between bug-fixing and regular development, with developers adding logging 2-3 times more frequently when fixing bugs. This pattern shows that developers add logging more frequently in complex distributed systems and actively increase it when problems occur.

Kafka, Flink, Elasticsearch, Cassandra, and Maven show smaller but consistent gaps, indicating moderate reactive logging strategies. Developers in these projects add logging during bug fixes but not as aggressively as in Hadoop and Tomcat. This suggests a pattern where the associations of logging with debugging are moderate.

Spring and Jenkins show no difference between bug-fixing and regular development contexts, indicating that developers in these projects show different logging patterns. These projects demonstrate lower logging addition rates, with different associations between logging and debugging activities.

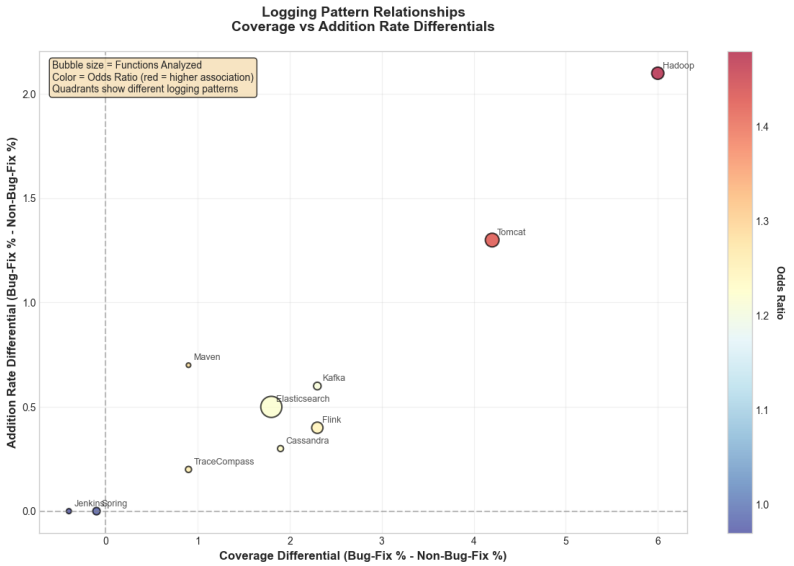

Figure 6.3 The relationship between coverage differentials and addition rate differentials.

Project-Specific Logging Associations

Figure 6.3 shows the relationship between the level of logging coverage between bug fixing and regular development (horizontal axis) and the level of logging that developers add during bug fixes (vertical axis). Projects in the upper-right area demonstrate strong logging-debugging relationships, while projects near the origin show weak relationships.

Hadoop and Tomcat occupy the upper-right position with large bubbles (indicating substantial sample sizes) and dark red coloring (indicating strong statistical associations). Their odds ratios above 1.4 mean that functions with logging are 40% more likely to be involved in bug fixes than functions without logging. This shows strong statistical associations between logging and debugging activities in these projects.

The middle region contains Kafka, Flink, Cassandra, Elasticsearch, and Maven with moderate positioning and yellow-orange coloring, indicating odds ratios around 1.2-1.3. This suggests logging provides some debugging value but less than in Hadoop and Tomcat.

Spring and Jenkins cluster near the origin with light blue coloring and odds ratios near 1.0, indicating no meaningful statistical relationship between logging and bug-fixing activities. This confirms weak statistical associations between logging and debugging activities in these projects.

Log Level Usage Pattern

Figure 6.4 shows how developers use different logging levels (info, debug, warn, error, trace, fatal) during bug-fixing versus regular development. Projects like Hadoop and Tomcat increase their use of error-level logging (red sections) during bug fixes, which makes sense because developers focus on critical problems when fixing bugs.

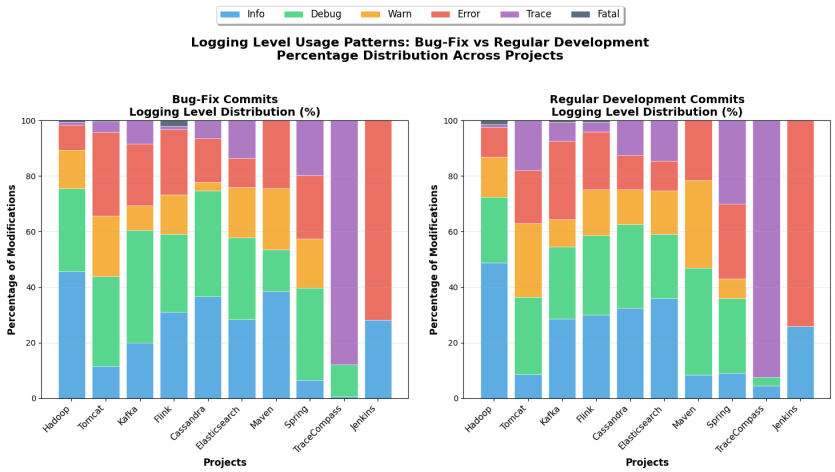

Figure 6.4 Logging level distributions.

Debug-level logging (green sections) remains important across both contexts but increases during bug fixes in projects like Hadoop, Tomcat, and Kafka. This suggests developers add detailed debugging information when investigating problems. Info-level logging (blue sections) stays relatively stable, indicating it serves documentation purposes rather than debugging.

Projects like Spring and Jenkins show minimal changes in logging level distributions between bug-fixing and regular development, reinforcing that logging does not serve a primary debugging function in these projects. TraceCompass shows unique patterns with substantial trace-level logging, reflecting its specialized monitoring purpose.

The consistent patterns within project groups confirm that the operational context determines how developers approach logging during debugging activities.

Logging Level Modification Patterns

Figure 6.5 demonstrates how developers modify different logging levels during bug-fixing versus regular development activities across projects. Hadoop shows the most pronounced modification behavior during bug fixes across multiple logging levels, with debug modifications at 2.7 per 100 functions during bug fixes compared to 1.1 during regular development, and info modifications at 4.1 versus 2.2 per 100 functions respectively.

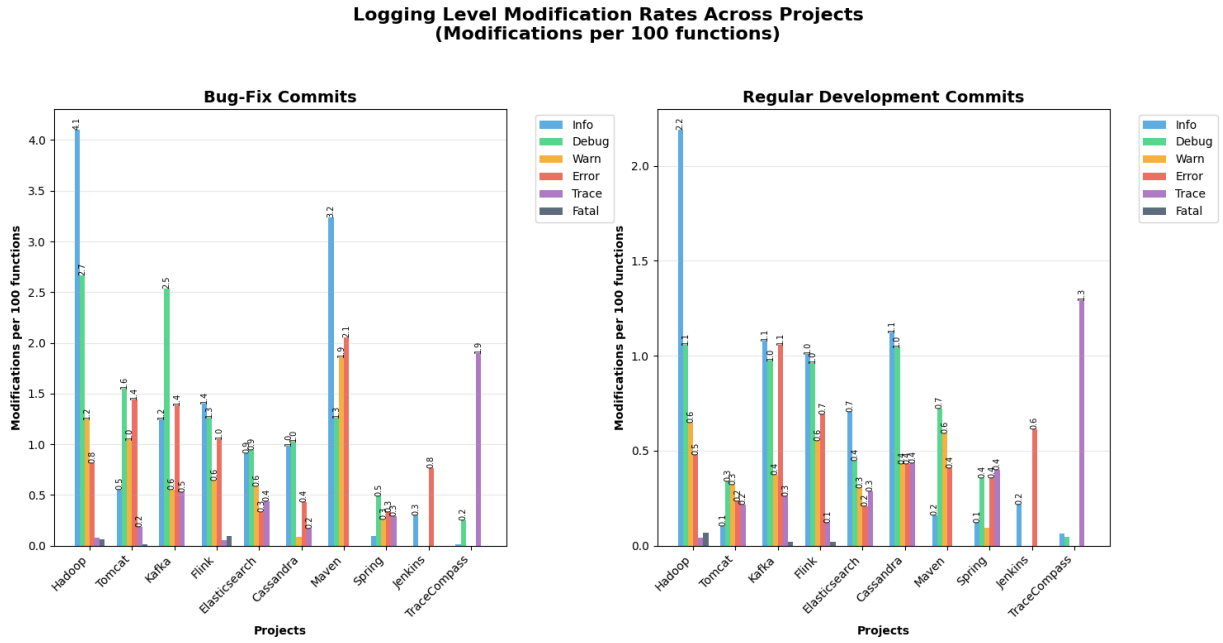

Figure 6.5 Logging level modification rate.

Tomcat exhibits substantial reactive patterns with debug modifications reaching 1.5 per 100 functions during bug fixes versus 0.3 during regular development, and error modifications at 1.4 compared to 0.2 per 100 functions. Kafka displays moderate increases with debug modifications at 2.5 during bug fixes versus 1.0 during regular development.

Flink and Elasticsearch show balanced modification patterns, with Flink demonstrating info modifications at 1.4 during bug fixes versus 1.0 during regular development, while Elasticsearch maintains similar rates across both contexts with debug modifications at approximately 0.9 per 100 functions in both cases.

Maven, Cassandra, and TraceCompass exhibit distinct patterns specific to their contexts. Maven shows elevated info modifications during bug fixes (3.2 vs 0.2 per 100 functions),

while TraceCompass demonstrates high trace-level modifications in both contexts (1.9 during bug fixes, 1.3 during regular development).

Spring and Jenkins demonstrate minimal differences between bug-fixing and regular development contexts. Spring shows nearly identical modification rates across all logging levels, with debug modifications at 0.5 per 100 functions in both contexts. Jenkins exhibits low overall modification activity with error modifications at 0.8 during bug fixes versus 0.6 during regular development.

When subtracting regular development rates from bug-fixing rates, projects like Hadoop, Tomcat, and Kafka show substantial positive differentials, particularly in debug-level logging, indicating reactive modification strategies where developers actively adjust logging when investigating problems.

6.5.3 RQ2: Statistical Associations Between Logging Presence and Bug Involvement

While RQ1 examined developer behavior during bug-fixing activities, RQ2 investigates logging effectiveness from a different perspective. RQ1 asked "What do developers DO with logging during bug fixes?" while RQ2 asks "Does having logging make functions more likely to be involved in bug fixes?" This shift from behavioral analysis to effectiveness analysis reveals whether existing logging practices actually correlate with bug involvement across different projects.

Statistical Associations Across Projects

Figure 6.6 demonstrates substantial variation in the statistical relationship between logging presence and bug involvement across projects. The odds ratios are calculated from 2x2 contingency tables comparing logged vs not logged functions in bug-fixing vs regular development contexts. Hadoop shows the strongest association with an odds ratio of 1.48, calculated as (3,398 logged bug-fix $\times$ 49,910 not logged regular) $\div$ (12,700 logged regular $\times$ 9,040 not logged bug-fix), indicating that functions with logging are 48% more likely to be involved in bug fixes than functions without logging.

Tomcat follows closely with an odds ratio of 1.43, while Kafka, Flink, Elasticsearch, Cassandra, Maven, and TraceCompass show moderate positive associations ranging from 1.21

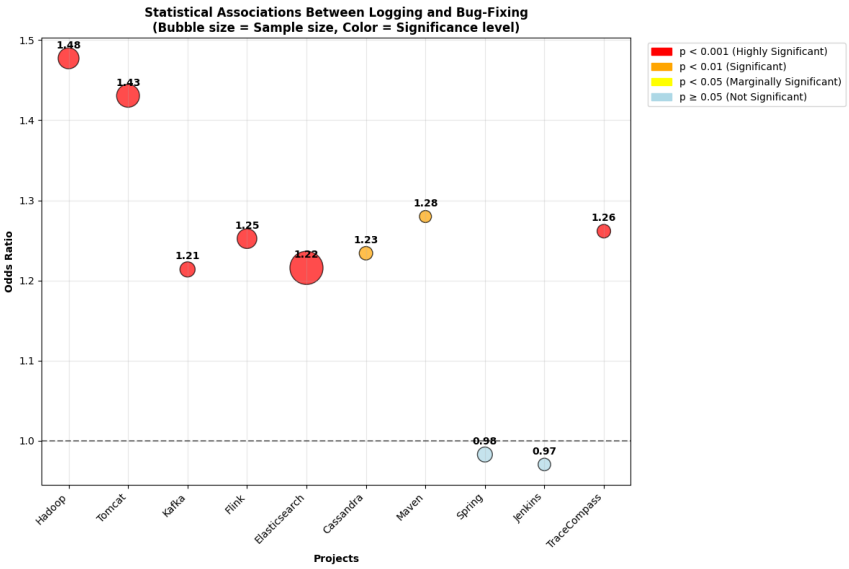

Figure 6.6 Correlation between logging and bug-fixing.

to 1.28. In contrast, Spring and Jenkins exhibit odds ratios near 1.0 (0.98 and 0.97 respectively), indicating no meaningful statistical relationship between logging presence and bug involvement. The bubble sizes in the figure reflect sample sizes, with Elasticsearch representing the largest analysis and providing confidence in the statistical robustness of these patterns.

Function Distribution Patterns

Figure 6.7 reveals distinct patterns in how logged and not logged functions are distributed across bug-fixing and regular development contexts through stacked percentage bars. Each bar represents 100% of a project's functions, divided into four categories: logged functions in bug fixes (red), logged functions in regular development (orange), not logged functions in bug fixes (blue), and not logged functions in regular development (green). The relative heights of these colored sections show the proportional distribution within each project.

Projects show varying proportions of logged versus not logged functions across both contexts. Tomcat shows a relatively high proportion of logged functions in bug fixes compared to other projects, while Spring demonstrates high overall function involvement in bug fixes regardless of logging status. Jenkins and Maven exhibit lower overall logging utilization, with logged functions representing smaller proportions of total project activity across both contexts.

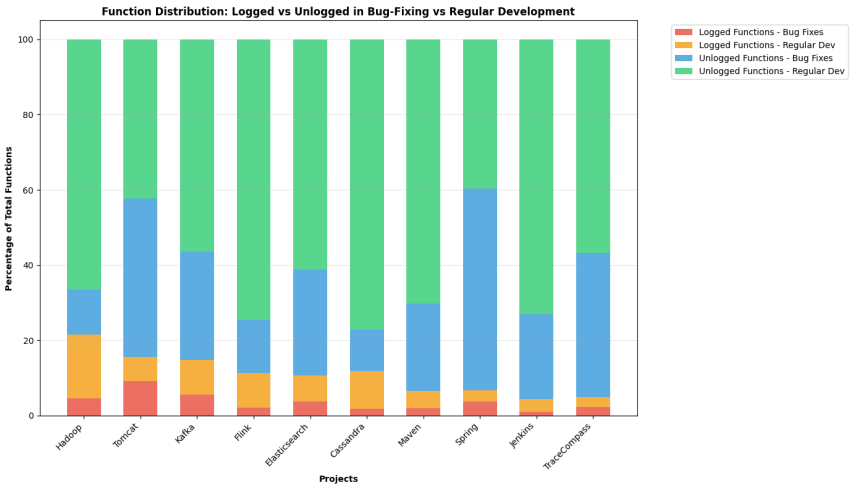

Figure 6.7 Distribution of logged and not logged function in bug-fixing and regular development activities.

Coverage vs Bug-Fix Rate Relationships

Figure 6.8 examines the relationship between logging coverage and overall project bug-fix rates through a scatter plot. Logging coverage is calculated as (total functions with logging ÷ total functions) × 100, while overall bug-fix rate is calculated as (total functions in bug fixes ÷ total functions) × 100. Hadoop appears in the lower right with approximately 21% logging coverage and 16.6% overall bug-fix rate, while Spring shows approximately 6.7% logging coverage with 57.5% overall bug-fix rate.

Tomcat demonstrates approximately 15.6% logging coverage with 51.3% overall bug-fix rate. The scatter plot reveals that projects with higher logging coverage do not necessarily have higher overall bug-fix rates. Jenkins shows low coverage (approximately 4. 4%) with low bug fix rates (23. 6%), while TraceCompass shows low coverage (approximately 5. 0%) but moderate bug fix rates (40. 6%). This distribution suggests that logging coverage and overall project bug rates operate independently.

Bug-Fix Likelihood Comparisons

Figure 6.9 provides direct comparison of bug-fix involvement rates between logged and not logged functions across projects. These rates are calculated as the percentage of each function type (logged/not logged) that appears in bug-fixing commits. For example, Hadoop's 21.1%

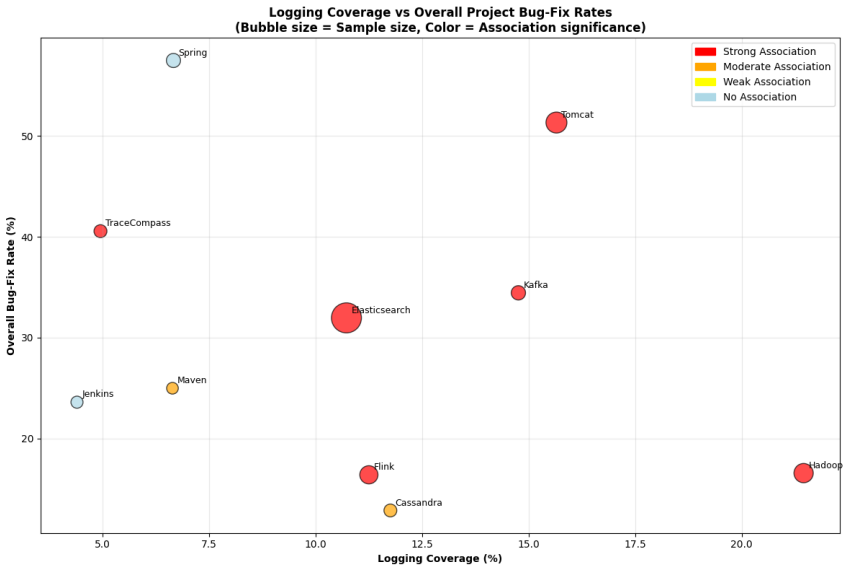

Figure 6.8 Correlation of logging coverage and bug-fixing activities.

rate for logged functions means that out of all functions with logging in Hadoop, 21.1% were involved in bug fixes, calculated as (3,398 logged functions in bug fixes ÷ 16,098 total logged functions) × 100.

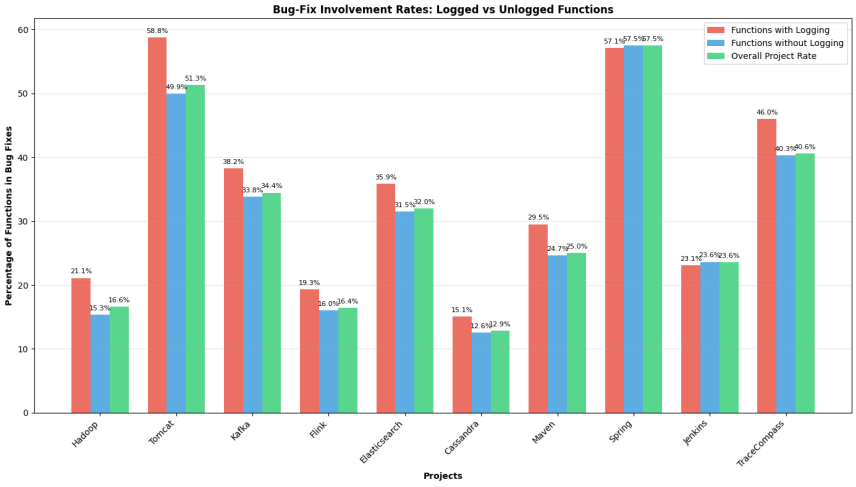

Figure 6.9 Percentage of logged and not-logged functions in bug-fixing commits.

Hadoop shows 21.1% of logged functions versus 15.3% of not logged functions involved in bug fixes, with an overall project rate of 16.6%. Tomcat exhibits 58.8% versus 49.9% involvement rates with an overall project rate of 51.3%. Kafka shows 38.2% versus 33.8% (overall 34.4%),

Flink demonstrates 19.3% versus 16.0% (overall 16.4%), and Elasticsearch shows 35.9% versus 31.5% (overall 32.0%).

Cassandra, Maven, and TraceCompass show smaller but consistent patterns favoring logged functions: Cassandra at 15.1% versus 12.6% (overall 12.9%), Maven at 29.5% versus 24.7% (overall 25.0%), and TraceCompass at 46.0% versus 40.3% (overall 40.6%). In contrast, Spring and Jenkins show minimal differences, with Spring demonstrating 57.1% versus 57.5% (overall 57.5%) and Jenkins showing 23.1% versus 23.6% (overall 23.6%), indicating no meaningful relationship between logging presence and bug involvement in these projects.

6.5.4 RQ3: Developer Perceptions vs. Observed Logging Practices

Our developer survey captured perspectives from 58 practitioners (27 industry professionals from Ericsson and 31 open-source contributors from our analyzed projects) to understand how stated logging intentions align with observed behaviors in our code analysis. This comparison reveals significant gaps between developer perceptions and actual practices, particularly regarding proactive versus reactive logging approaches.

Perception-Reality Gap in Logging Strategies

Figure 6.10(A) shows how developers from different contexts perceive their logging strategies. Industry developers claim to use "a mix of both" proactive and reactive approaches (74.1%), presenting themselves as balanced in their logging decisions. In contrast, only 32.3% of open-source developers make similar claims, with equal proportions (32.3%) admitting to "proactive" approaches and others acknowledging reactive patterns.

Our code analysis reveals reactive logging patterns: developers add logging 1.2-3.8x more frequently during bug fixes than regular development, and functions with logging are 1.21-1.48x more likely to be involved in bug-fixing activities. When developers encounter problems, they add logging to aid in debugging, rather than anticipating issues.

Open-source survey responses align better with observed behavior. Open-source developers are less likely to claim balanced approaches, which matches the reactive patterns observed in the GitHub projects they contribute to. This suggests a perception-reality gap where developers, particularly those in industry contexts, may believe they're being more strategic in their logging decisions than their actual practices demonstrate.

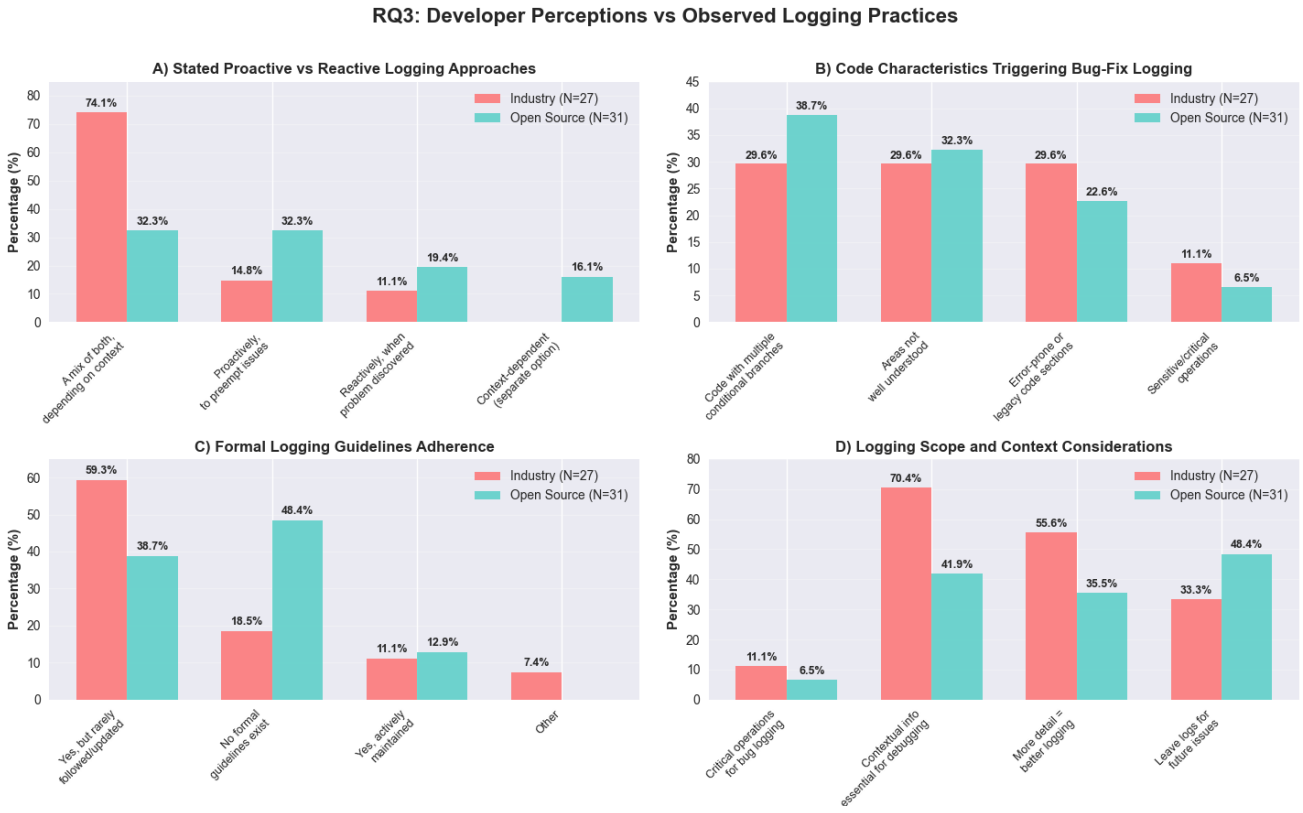

Figure 6.10 Survey results for both industry and open-source participants.

Code Complexity vs. Legacy Focus

Figure 6.10(B) shows differences in what triggers developers to add logging during bug-fixing activities. Industry developers focus more on "error-prone or legacy code sections" (29.6% vs 22.6%), reflecting their need to deal with existing codebases and maintenance burdens. This pattern suggests that industry developers often encounter bugs in older, less well-understood parts of systems.

Open-source developers focus more on "code with multiple conditional branches" (38.7% vs 29.6%) and "areas not well understood" (32.3% vs 29.6%). This emphasis on structural complexity aligns with our RQ1 and RQ2 findings, which show that the effectiveness of the log varies between project domains and complexity levels. Projects handling complex operations and distributed systems (such as Hadoop and Tomcat) show stronger logging-debugging relationships, suggesting that complexity-driven logging decisions may be more effective than legacy code-driven ones.

This difference in trigger patterns may explain why our analysis shows domain-dependent

effectiveness: developers who focus on code complexity (multiple branches, unclear logic) rather than legacy status may place logging more effectively for debugging purposes.

Organic vs. Systematic Development

Figure 6.10(C) shows patterns regarding formal logging guidelines. Open-source projects are more likely to operate without formal logging guidelines (48.4% vs 18.5% in industry), yet these developers show better alignment between their stated practices and observed behaviors in our code analysis.

Industry environments show a higher adoption of formal guidelines, with 59.3% having guidelines that are "rarely followed/updated" compared to 38.7% in open-source contexts. Additionally, 11.1% of industry teams selected "actively maintained" guidelines versus 12.9% in open-source projects.

This pattern suggests that formal guidelines may not correlate with effective logging practices. The open-source developers in our study, despite having fewer formal guidelines, show more accurate self-assessment of their reactive logging approaches. This indicates that effective logging practices emerge from direct problem-solving experience and project-specific needs rather than from standardized processes.

The implication for software practice is that organizations investing in formal logging guidelines should focus on ensuring these guidelines reflect actual debugging needs and are actively maintained, rather than assuming that formal documentation automatically leads to better practices.

Context and Scope Considerations

Figure 6.10(D) shows patterns in how developers approach logging scope and contextual information. Both industry and open-source developers show low consideration for "sensitive/critical operations" when deciding where to add bug-triggered logging (11.1% industry vs 6.5% open-source). This finding suggests that security and critical operation logging may be handled separately from general debugging logging practices.

Industry developers show higher prioritization of contextual information as "essential to resolving issues quickly" (70.4% vs 41.9%), while open-source developers are more likely to "leave logs for future issues" after bug fixes (48.4% vs 33.3%). This reflects different op-

erational contexts where industry developers need immediate problem resolution with rich context, while open-source developers build logging infrastructure for future debugging scenarios.

The separation of critical operations from general debugging logging could explain some of the domain-dependent effectiveness patterns observed in our code analysis. Infrastructure projects like Hadoop and Tomcat, which handle critical distributed operations, show stronger logging-debugging relationships (odds ratios 1.43-1.48). However, if critical operations logging is managed through separate processes rather than integrated with general debugging practices, this could limit the overall effectiveness of logging for comprehensive system understanding.

Implications for Logging Practice

The survey data shows a pattern suggesting that effective logging practices may emerge from direct problem-solving experience rather than from formal systematic approaches. Open-source developers show several characteristics supporting this hypothesis: they are more likely to operate without formal guidelines (48.4% vs 18.5%), show better self-awareness of their reactive patterns, and recognize context-dependency in their approaches.

This pattern aligns with our code analysis findings showing reactive logging behaviors across GitHub projects. Developers encounter problems, add logging to solve them, and build understanding of where logging provides debugging value. This experiential learning may produce more effective logging placement than systematic approaches that do not account for project-specific debugging needs.

The implication is that organizations seeking to improve logging effectiveness should consider approaches that capture and codify the learning from debugging experiences rather than imposing top-down systematic guidelines that may not reflect actual debugging patterns.

These RQ3 findings provide context for interpreting our RQ1 and RQ2 results: the domain-dependent logging associations we observed may result from experience-driven logging practices that vary across different project contexts and developer communities, rather than from consistent application of formal logging strategies.

6.6 Discussion

Our empirical analysis of over 572,000 functions across ten Java projects, combined with insights from 58 developers, reveals patterns in logging effectiveness that challenge conventional assumptions about logging practices. This section synthesizes our findings to provide implications for software practitioners and the research community.

6.6.1 Domain-Dependent Logging Associations

Our analysis reveals that logging associations with bug involvement vary substantially across software domains, with infrastructure projects (Hadoop, Tomcat, Kafka) demonstrating stronger statistical associations between logging and bug involvement (odds ratios 1.21-1.48) while framework projects (Spring, Jenkins) show weaker associations (odds ratios near 1.0). This suggests that logging patterns may need to consider project characteristics rather than applying uniform approaches.

For Infrastructure and Distributed Systems: In the infrastructure projects we studied, developers add logging 2-3.8 times more frequently during bug fixes, and functions with logging are 21-48% more likely to be involved in bug resolution activities. This pattern suggests that for projects with similar characteristics—complex distributed operations, unpredictable failure modes—logging shows substantial statistical associations with debugging activities.

For Framework and Utility Projects: Our analysis of framework projects shows weaker statistical relationships between logging presence and bug involvement (odds ratios near 1.0). This pattern suggests that for projects with similar characteristics, extensive logging may yield more limited returns in terms of bug resolution support. The reasons for this difference could include different debugging approaches, different types of problems, or different operational environments.

For Java Development Organizations: The domain-dependent patterns we observed in Java projects suggest that the statistical associations between logging and bug involvement vary based on project characteristics rather than being uniform across contexts. However, understanding whether these patterns apply in specific organizational contexts would require careful assessment of individual project characteristics and development environments. These findings provide a baseline for Java development that organizations can use as a reference point for examining their own contexts.

The differences we observed between infrastructure and framework projects may reflect fun-

damental differences in testing accessibility and codebase characteristics. Framework projects like Spring and Jenkins are extensively exercised by the diverse applications that depend on them, creating implicit testing that may expose bugs through regular usage patterns. This extensive application-level testing could provide alternative mechanisms for bug detection and diagnosis, reducing reliance on internal logging for debugging purposes.

In contrast, infrastructure projects operate with more diverse and conditionally executed code paths. Projects like Hadoop and Tomcat must handle various drivers, optional features, and configuration-dependent execution paths that receive limited coverage through normal testing scenarios. When bugs occur in these rarely-exercised code paths, developers have fewer alternative debugging approaches and must rely on targeted logging to understand system state at failure points.

6.6.2 Reactive Logging Patterns and Developer Perceptions

Our code analysis reveals predominantly reactive logging patterns across the studied projects, where developers add logging when encountering problems rather than proactively planning comprehensive coverage. However, our survey indicates that 74.1% of industry developers believe they follow *balanced* approaches, suggesting a gap between perceived and actual practices.

Understanding Actual vs. Perceived Practices: This perception-reality gap indicates that developers may not accurately assess their own logging practices. In our studied projects, the predominant pattern is reactive logging, yet many developers believe they follow more strategic approaches. This disconnect suggests that organizations and researchers may be working with incomplete understanding of how logging actually occurs in practice.

Implications for Self-Assessment and Training: The gap between perceived and actual practices reveals that developers may not accurately recognize their own logging behaviors. Rather than assuming developers know how they approach logging, our findings suggest that actual logging patterns differ substantially from perceived patterns, indicating that current understanding of developer logging practices may be incomplete.

Organizational Policy Considerations: Many organizations may have policies or expectations based on the assumption that developers follow "balanced" or proactive logging approaches. Our findings reveal that actual practice is predominantly reactive, which indicates a potential mismatch between organizational expectations and developer reality. This

suggests that logging policies may be based on inaccurate assumptions about how logging actually occurs in their development processes.

Research Community Insights: For researchers, this finding reveals that studies based on developer self-reporting of logging practices may not accurately capture actual behavior. The significant gap we observed between stated approaches (74.1% claiming balanced) and observed patterns (predominantly reactive) suggests that empirical analysis of code changes may provide more accurate insights into logging practices than survey-based approaches alone. This has implications for how logging research is conducted and how findings are interpreted.

6.6.3 Implications for Research Advancement

Understanding Reactive Logging: Current logging research often focuses on predicting optimal logging placement proactively, but our findings from Java projects show that reactive logging is the predominant pattern in open-source development practice. Research investigating when and why reactive logging shows strong statistical associations with debugging activities in Java and other programming contexts, and how these patterns emerge across different development environments, could better align with actual developer practices across diverse programming ecosystems.

Empirical Validation of Logging Tools: Many logging tools and techniques are developed based on theoretical assumptions about optimal logging practices. Our findings suggest that these assumptions may not reflect actual developer behavior or needs. Research that empirically validates logging tools against observed debugging patterns and outcomes could better align tool design with actual practices.

6.6.4 Practical Considerations for Implementation

Our findings suggest several practical considerations for organizations and development teams, though implementation should be carefully adapted to specific contexts.

Context-Aware Logging Strategies: The domain-dependent association patterns we observed suggest that logging patterns vary based on project-specific characteristics such as operational complexity, failure modes, and debugging requirements. However, understanding these patterns would require careful analysis of individual project contexts.

Realistic Practice Assessment: The perception-reality gap we identified reveals that or-

ganizations may have incomplete understanding of their actual logging practices when relying on assumptions or self-reports. This suggests that analyzing actual code changes, debugging workflows, or incident response patterns could provide more accurate insights into how logging occurs in practice.

Resource Allocation Decisions: Our findings from Java projects suggest that logging associations with bug involvement vary significantly across different types of projects within this ecosystem. Organizations developing Java applications and making resource allocation decisions about logging infrastructure, training, or tools may want to examine whether similar patterns exist in their specific contexts, using our findings as a baseline for comparison rather than applying uniform approaches. These patterns warrant investigation in other programming contexts to establish broader applicability.

Developer Support Systems: The predominantly reactive logging patterns we observed suggest that development tools and processes designed around proactive logging assumptions may not align with actual developer workflows. This indicates that tools facilitating logging during debugging sessions or processes that capture logging patterns during actual debugging activities could better match observed practices.

These implications are based on our analysis of ten Java projects and survey responses from 58 developers. The applicability of these findings to other contexts, programming languages, and development environments should be carefully considered, as different contexts may exhibit different patterns and relationships.

While our analysis reveals consistent domain-dependent patterns in logging associations with bug involvement, the underlying reasons for these differences require further investigation. The stronger associations we observed in infrastructure projects (Hadoop, Tomcat, Kafka) compared to framework projects (Spring, Jenkins) represent an empirical pattern that warrants theoretical development in future research. Our findings do not establish why different software domains exhibit these distinct logging patterns, but they provide clear evidence that such differences exist and are statistically significant across multiple projects. Understanding the mechanisms that drive these domain-dependent patterns could provide important insights into how software characteristics relate to debugging practices and logging strategies. These empirical findings establish the foundation for future research to investigate what causes these patterns and whether they extend to other software domains and programming contexts.

6.7 Threats to Validity

This section discusses potential threats to the validity of our findings and the specific measures we took to mitigate them.

6.7.1 Internal Validity

Internal validity concerns whether the relationships we observed between logging practices and bug-fixing activities reflect true associational relationships rather than artifacts of our methodology.

Commit Classification Accuracy: Our automated bug-fixing commit classification could potentially misclassify commits that use unconventional terminology or lack explicit problem indicators. To mitigate this threat, we developed a comprehensive hierarchical four-strategy approach (GitHub issue references, revert patterns, Java-specific error patterns, and action-problem keyword combinations) and validated our classification through manual verification of 100 commits per project, achieving 91% accuracy across all projects. We used a statistically determined sample size (90% confidence level, 10% margin of error) to ensure reliable validation.

Function Identification Completeness: Complex refactoring operations, file renaming, or non-standard code formatting could cause our diff parsing system to miss some modified functions. We addressed this by implementing a comprehensive three-tier search strategy: direct change line analysis, context line analysis, and backward file scanning to maximize function capture. When ambiguous cases occurred, we excluded them rather than making assumptions, ensuring that our analysis includes only clearly identifiable function modifications.

Logging Detection Accuracy: Projects using non-standard logging approaches or custom utilities might not be captured by our detection system. To minimize this threat, we designed our AST-based approach to recognize multiple logging patterns across common Java frameworks and examined import statements to identify project-specific logging frameworks. We focused on six standard logging levels (info, debug, trace, warn, error, fatal) that are widely adopted across Java projects, ensuring broad applicability of our detection approach.

Temporal Relationship Interpretation: Our statistical associations cannot establish causal relationships between logging and debugging effectiveness. We addressed this limitation by clearly presenting our findings as associational patterns rather than causal claims, and by triangulating our code analysis with developer survey responses to gain insights into

the decision-making processes behind observed patterns. We explicitly acknowledge that our reactive logging patterns suggest much logging is added during debugging activities rather than preventing bugs.

6.7.2 External Validity

External validity concerns the generalizability of our findings beyond the specific projects, contexts, and populations we studied.

Programming Language Scope: Our focus on Java projects could limit generalizability to other programming languages with different characteristics. We mitigated this by selecting Java specifically because of its widespread adoption in both enterprise and open-source development, mature logging ecosystem, and consistent frameworks across projects, making it representative of mainstream software development practices. Our findings provide a solid foundation for Java development contexts and a baseline for future cross-language studies.

Open-Source Context Limitation: Analyzing only GitHub projects might not represent industrial development practices or proprietary codebases. To address this limitation, we complemented our open-source code analysis with a targeted survey of industry professionals from Ericsson, enabling us to compare developer perspectives across both contexts. This mixed-methods approach revealed important differences between industry and open-source developer practices, providing insights that neither approach alone could capture.

Project Selection Approach: Our purposeful selection of projects based on popularity, active maintenance, and domain diversity could bias toward successful projects with mature practices. We mitigated this by explicitly designing our selection criteria to ensure analysis of well-established, actively developed projects that represent different software domains (infrastructure, frameworks, tools). This approach provides insights into logging practices in production-quality systems while acknowledging that patterns may differ in smaller or newer projects.

Developer Survey Response Limitations: Our 5.2% response rate among open-source contributors could introduce response bias, and our industry sample from a single organization might not represent all industries. We addressed these limitations by achieving balanced representation (27 industry, 31 open-source respondents) and by designing our survey questions to capture diverse perspectives on logging practices. The response rate is typical for unsolicited developer surveys, and our mixed-methods approach allows us to identify discrepancies between stated practices and observed behaviors that single-method studies cannot

detect.

Scope and Generalizability Positioning: Our focus on Java projects and GitHub repositories establishes baseline patterns within a specific but important segment of software development. While this limits direct generalizability to other programming languages and development contexts, it provides a foundational empirical understanding of logging patterns in a major programming ecosystem that can inform future cross-language studies. The Java ecosystem represents a substantial portion of enterprise and open-source development, making our findings relevant as a starting point for broader investigations. Future research should examine whether the domain-dependent patterns we identified in Java development occur across other programming contexts, using our findings as a reference point for comparison.

6.7.3 Construct Validity

Construct validity concerns whether our measurements and definitions accurately capture the intended concepts of logging effectiveness, bug-fixing activities, and developer practices.

Bug-Fixing Definition Limitations: Relying on textual indicators in commit messages and issue labels might miss some debugging activities or misclassify improvement work as bug fixes. We mitigated this by developing a comprehensive multi-pattern detection system that captures various ways developers communicate bug-fixing activities, from explicit GitHub issue references to technical error patterns specific to Java development. Our 91% validation accuracy demonstrates that our approach successfully identifies developer-expressed bug-fixing intent across diverse communication styles.

Logging Effectiveness Measurement: Using statistical associations as a proxy for actual debugging value might not capture the full benefit logging provides during problem-solving activities. We addressed this limitation by combining quantitative code analysis with qualitative survey insights to understand both behavioral patterns and developer perspectives on logging value. This dual approach allows us to identify not only where logging is associated with bug involvement but also how developers perceive and use logging during debugging activities.

Domain Classification Validity: Categorizing projects into broad domains might oversimplify complex software systems that serve multiple purposes. We addressed this by basing

our classification on primary project functions and intended use cases while acknowledging that projects may evolve or serve hybrid roles. Our analysis focuses on identifying patterns within our classifications while recognizing that domain boundaries are not absolute, and we present our findings as applicable to projects with similar characteristics rather than universal principles.

6.8 Conclusion and Future Work

This study set out to investigate the relationship between logging practices and bug resolution in software development through empirical analysis of real-world development activities. Our research questioned fundamental assumptions about logging effectiveness by examining whether logging actually supports debugging activities as commonly believed, and how developer practices align with stated intentions regarding logging strategies.

Our findings provide substantial evidence that challenges conventional wisdom about logging practices in several key ways. Through analysis of over 572,000 functions across ten diverse Java projects, we demonstrated that logging associations varies across software domains, with infrastructure projects (Hadoop, Tomcat, Kafka) showing strong statistical associations between logging and bug involvement (odds ratios 1.21-1.48) while framework projects (Spring, Jenkins) show minimal associations (odds ratios near 1.0). Our mixed-methods approach revealed that developers follow predominantly reactive logging patterns—adding logging when problems occur rather than proactively planning comprehensive coverage—yet 74.1% of industry developers believe they follow "balanced" approaches, indicating a significant gap between perceived and actual practices. Additionally, our survey of 58 developers from both industry and open-source communities uncovered systematic differences in how developers approach logging decisions, with open-source developers showing better alignment between their stated practices and observed behaviors.

These findings challenge how the software development community thinks about logging practices and their relationship to debugging activities. For decades, the industry has operated under assumptions that comprehensive logging is universally beneficial and that developers should plan logging proactively. Our evidence suggests these assumptions may be incorrect—that logging associations with bug involvement depend heavily on project context, and that reactive logging may be not only common but the predominant pattern across many development scenarios. This has immediate implications for how organizations allocate resources, design development processes, and evaluate developer practices. Rather than

mandating uniform logging standards across all projects, our findings suggest that logging associations vary across domain-specific characteristics. For the research community, our results indicate that current approaches to logging research may be fundamentally misdirected, as many studies assume universal best practices or rely on self-reported developer behaviors that do not align with actual practices.

Our work provides a large-scale empirical foundation for understanding when and how logging shows statistical associations with bug resolution activities. By combining quantitative analysis of actual development behaviors with qualitative insights from developer perspectives, we offer evidence-based insights that move beyond theoretical assumptions about optimal logging practices. The domain-dependent association patterns we identified suggest that the software engineering community should abandon one-size-fits-all approaches to logging and instead examine context-aware patterns that align with empirical evidence of logging associations with debugging activities. Our findings also highlight the importance of understanding actual developer practices rather than relying on assumptions about how development activities occur, as the substantial perception-reality gaps we observed suggest that much of what we think we know about developer behavior may be inaccurate.

The implications of this research extend far beyond logging practices to broader questions about how software engineering knowledge is developed and validated. Our study demonstrates the value of empirical analysis over theoretical assumptions, the importance of context-dependent rather than universal solutions, and the need for research that examines actual developer behaviors rather than idealized practices. As software systems become increasingly complex and diverse, evidence-based approaches to development practices become more critical for understanding actual software engineering patterns. This research establishes a foundation for such evidence-based understanding of logging practices while highlighting the importance of validating commonly accepted assumptions through rigorous empirical investigation.

This research opens several promising directions for advancing the understanding of logging associations with debugging activities and software debugging practices. Cross-language and cross-context validation represents a critical next step for establishing the generalizability of our domain-dependent association patterns. Future studies should extend our Java-focused analysis to other programming languages with different error handling paradigms, memory management approaches, and logging frameworks to determine whether the infrastructure

versus framework association differences we observed reflect general principles or language-specific characteristics. Additionally, systematic analysis of industrial logging practices across different domains would validate our findings in commercial development contexts and potentially reveal additional patterns that our GitHub-based analysis could not capture.

Longitudinal outcome studies offer another valuable research direction that could provide direct evidence of logging's relationship to debugging outcomes. Rather than examining statistical associations between logging presence and bug involvement, future research could track actual debugging sessions, measure problem resolution times, and assess how different logging patterns relate to developer problem-solving outcomes across various project contexts. Such studies could also investigate the long-term maintenance implications of different logging approaches, examining whether reactive logging patterns result in different debugging outcomes over time compared to comprehensive upfront planning approaches. These longitudinal investigations would provide the causal evidence that our correlational study cannot establish, offering definitive guidance on when and how logging relates to software maintenance outcomes across different development scenarios.

Acknowledgment

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), AMD, Ciena, Ericsson, and EfficiOS for funding this research project.

CHAPTER 7 CONCLUSION

This chapter presents the concluding discussion of this thesis on cost-aware software performance analysis. The chapter first presents a summary of the contributions achieved through the three research papers, followed by the limitations that we identified in our research, and concludes with the possible future research directions that build upon the foundations established in this work.

7.1 Summary of Works

The first contribution, "Balancing Costs and Insights: A Framework for Cost-Aware Software Instrumentation," established the theoretical and practical foundations for systematic instrumentation decision-making by developing the first comprehensive multi-dimensional cost taxonomy encompassing twelve distinct overhead categories across intrinsic versus extrinsic, operational versus non-operational, and immediate versus deferred dimensions. The research achieved validation results with correlation values of 0.873 for CPU-intensive applications, 0.792 for IO-intensive applications, and 0.731 for network-intensive applications between predicted and actual Coefficient of Variation (CoV) measurements, demonstrating that the metric-based approach of our framework accurately predicts execution time variability across diverse resource utilization patterns, as the Coefficient of Variation directly measures execution time unpredictability, the characteristic that indicates functions requiring performance monitoring attention. Evaluation across three applications—SPEC CPU benchmark, Redis database, and Apache HTTPD web server—showed that the optimization framework consistently identified functions ranking within the top 12-27% based on actual execution time variability while respecting specified cost constraints. Functions with high execution time variability indicate performance bottlenecks, resource contention, or state-dependent behavior, making them critical monitoring targets for effective performance diagnosis. This work benefits the software engineering community by providing both a structured mathematical formulation that transforms ad-hoc instrumentation decisions into tractable optimization problems and practical methodologies that enable developers to automatically balance monitoring effectiveness against multiple resource constraints, advancing the field toward systematic, cost-aware monitoring practices that scale to production environments.

The second contribution, "Automatic Reduction of Execution Trace Data Volume Using Gradient Boosting in Large-Scale Microservice Systems," addressed the critical scalability challenge of trace data analysis by developing a two-phase machine learning approach that

achieved over 69% reduction in trace data volume while maintaining or improving modeling accuracy. Using a comprehensive production dataset from Alibaba spanning 13 days across 40,000+ bare-metal nodes and 470,000+ containers with 28,000+ microservices, the research demonstrated RMSE scores of 0.02 for CPU utilization and 0.13 for memory utilization compared to 0.08 and 0.14 achieved with the full dataset, representing both significant data reduction and improved prediction accuracy. The analysis revealed that the most influential features for resource utilization modeling were concentrated in inter-service communication, memory access patterns, and database interactions, validating that targeted data collection focusing on critical architectural aspects can substantially decrease monitoring overhead without sacrificing analytical capability. This research provides the software monitoring community with empirically validated strategies for managing the data deluge problem that has hindered comprehensive monitoring adoption, enabling organizations to maintain effective performance analysis while operating within realistic storage, bandwidth, and processing constraints characteristic of modern distributed systems.

The third contribution, "Logging Practices in Software Bug Resolution: An Empirical Study," established essential empirical foundations through comprehensive analysis of over 572,000 functions across ten major Java projects combined with surveys of 58 professional developers, achieving 91% accuracy in automated commit classification validation. The research revealed significant domain-dependent variations with infrastructure projects (Hadoop, Tomcat, Kafka) demonstrating strong statistical associations between logging and bug involvement (odds ratios 1.21-1.48) while framework projects (Spring, Jenkins) showed minimal associations (odds ratios near 1.0), and discovered that developers add logging 1.2-3.8 times more frequently during bug fixes than regular development. The study identified a substantial perception-reality gap where 74.1% of industry developers claim balanced logging approaches while code analysis revealed predominantly reactive patterns, and documented logging coverage variations ranging from 3.9% in CI/CD platforms to 25.5% in infrastructure projects. This research benefits both the academic community and software practitioners by providing the first large-scale empirical evidence that challenges common assumptions about logging effectiveness, establishing evidence-based baselines that inform cost-aware instrumentation decisions across different software domains, and creating systematic foundations for future causal studies that can track actual debugging outcomes and measure the direct impact of different logging strategies on problem resolution efficiency in real-world development contexts.

These three contributions converge toward the main goal of this thesis: reducing the costs associated with software performance analysis while maintaining monitoring effectiveness. The first contribution minimizes instrumentation costs by formulating placement decisions as

multi-constraint optimization problems. The second contribution directly addresses analysis costs by reducing the volume of trace data while preserving the accuracy of the modeling. The third contribution analyzes the associations between logging practices and bug involvement across different domains, providing cost-aware foundations for instrumentation decisions by identifying where logging investment is most likely to provide debugging value.

7.2 Limitations

The cost-aware instrumentation framework presented in Chapter 4 was developed and evaluated within the context of C programming language applications, which reflects the practical need to focus on a specific language ecosystem to develop precise static analysis capabilities and validate the optimization approach thoroughly. Although this choice enables detailed analysis of code characteristics and accurate cost measurement, it naturally raises questions about applicability to other programming languages with different runtime behaviors, memory management approaches, and instrumentation mechanisms. However, the mathematical optimization formulation and multi-dimensional cost taxonomy represent language-agnostic concepts that provide theoretical foundations for extending the approach to other programming contexts. The static analysis components would require adaptation to different language syntax and runtime characteristics, but the core optimization methodology and cost categorization framework remain generalizable.

Similarly, while the evaluation of our cost-aware framework was conducted across three applications representing different computational patterns, the systematic methodology demonstrated can guide similar evaluations in additional application domains by adapting the metric extraction process to capture domain-specific performance characteristics. The framework addresses execution time overhead and storage costs as representative examples of the broader cost spectrum, which provides validation of the multi-constraint optimization approach while acknowledging that production deployments may encounter additional cost dimensions. This focus was deliberate, enabling the validation of the mathematical framework with measurable cost types, but the design explicitly accommodates extension to other overhead categories such as energy consumption, security costs, or maintenance burden as measurement techniques for these dimensions mature. The practical deployment of the framework requires initial profiling to determine application resource utilization patterns, which represents an upfront investment that organizations must weigh against the long-term benefits of systematic instrumentation decisions.

The trace data volume reduction approach was developed using microservice architectures and validated through Alibaba's comprehensive production dataset, which provides substan-

tial real-world evidence but raises questions about generalizability to other distributed system architectures and organizational contexts. The microservice focus reflects the current industry trend toward service-oriented architectures where trace data volume problems are most acute, yet the underlying machine learning methodology of identifying critical features for resource prediction represents a more general approach that could be adapted to other distributed system types. The concentration on CPU and memory utilization as target metrics was motivated by their universal importance across computing environments, though this leaves other potentially critical performance indicators such as network latency patterns or application-specific business metrics unaddressed. The gradient boosting framework provides the flexibility to incorporate additional metrics as needed, but such extensions would require additional validation to ensure the feature selection approach remains effective across different performance dimensions.

The empirical study of logging practices focuses on Java projects and GitHub repositories, which represents both a strength in providing depth within a specific ecosystem and a limitation in terms of broader applicability. Java was selected due to its widespread adoption in enterprise development and mature logging frameworks, enabling meaningful analysis of established logging patterns, but this choice naturally limits direct generalizability to programming languages with different error handling paradigms or logging conventions. The GitHub-based analysis provides access to complete development histories and large-scale data, yet may not fully capture the nuances of proprietary enterprise development practices where different organizational constraints and development processes might influence logging decisions.

The research establishes associational patterns between logging presence and bug involvement rather than direct causal relationships, which represents both a methodological limitation and a necessary starting point for this line of research. Demonstrating causality would require controlled experiments or longitudinal studies tracking actual debugging outcomes, which present significant practical challenges when conducted at the scale necessary for statistical significance. The associational findings provide essential empirical foundations that enable future research to design targeted causal studies with appropriate experimental controls. The commit classification methodology, while achieving 91% accuracy through systematic validation, relies on textual indicators in commit messages and issue labels, which may not capture all bug-fixing activities, particularly those following non-standard communication patterns or occurring in organizations with different documentation practices.

7.3 Future Research

Cost-aware performance analysis has evolved through the contributions of this thesis, but several important research directions remain unexplored. One future direction is to develop adaptive monitoring systems that can adjust instrumentation strategies based on changing conditions without manual intervention. Current approaches, including the framework developed in this paper, require a configuration in advance and remain static during execution. Future work could explore systems that monitor their overhead and automatically adjust instrumentation density when resource constraints change or when workload patterns shift. Another direction involves standardizing cost measurement methodologies across different computing environments. While this thesis established a multi-dimensional cost taxonomy, the field lacks consistent measurement approaches that would enable meaningful comparison between different instrumentation strategies across organizations and platforms.

The cost-aware instrumentation framework provides a foundation that can be extended in several practical directions. The most immediate extension involves implementing the approach for additional programming languages, particularly Java and Python, where the static analysis techniques would need adaptation but the mathematical optimization core remains applicable. More challenging is developing dynamic instrumentation adjustment capabilities, where the system could modify its monitoring configuration during runtime based on changing performance characteristics or resource availability. This would require solving the technical challenge of safely adding or removing instrumentation points without system disruption. Integration with continuous integration systems represents another valuable direction, where the framework could analyze code changes and automatically suggest instrumentation updates as applications evolve. Finally, expanding the cost measurement approach to include energy consumption and security overhead would address two increasingly important concerns in modern software deployment.

The trace data volume reduction methodology opens several research paths focused on broader applicability and real-time operation. Extending the approach beyond microservices to other distributed system architectures such as serverless computing or edge computing environments would validate the generality of the machine learning approach. The current methodology requires offline analysis to identify important features, but developing real-time adaptive feature selection could enable systems to adjust their data collection strategy as application behavior changes. Another promising direction involves applying the approach to additional performance metrics beyond CPU and memory utilization, such as network latency or storage throughput, which would require developing new feature extraction techniques specific to these metrics. Cross-organizational validation studies would help establish

whether the patterns identified in the Alibaba dataset generalize to other computing environments and application types.

The empirical study of logging practices reveals several gaps that future research should address. The most important next step involves conducting longitudinal studies that track actual debugging outcomes over time, measuring whether logging presence actually reduces bug resolution time or improves developer productivity. This would require developing systems that can automatically correlate logging decisions with debugging success rates in real development environments. Expanding the empirical analysis to other programming languages would establish whether the domain-dependent patterns observed in Java projects represent universal software engineering phenomena or language-specific characteristics. Industrial partnership studies could validate the findings in commercial development environments where different organizational constraints and development processes could influence the effectiveness of logging. Another valuable direction involves developing automated log-recommendation systems that apply the empirical patterns discovered in this research, providing developers with evidence-based suggestions about where and how to add logs based on code characteristics and project domain.

These research directions build from the foundations established in this thesis, while addressing the practical needs of modern software development. The convergence of optimization-based placement decisions, intelligent data management, and empirical understanding of logging effectiveness creates opportunities for developing monitoring systems that automatically balance comprehensive observability against operational constraints. Future work pursuing these directions could transform software monitoring from ad-hoc practices toward systematic, evidence-based approaches that scale to the complexity of modern systems.

REFERENCES

[1] H. Li, W. Shang, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2858–2873, 2020.

[2] D. G. Reichelt, L. Bulej, R. Jung, and A. Van Hoorn, "Overhead comparison of instrumentation frameworks," in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, 2024, pp. 249–256.

[3] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[4] K. Guan and O. Legunsen, "An in-depth study of runtime verification overheads during software testing," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1798–1810.

[5] H. M. Salkin and C. A. De Kluyver, "The knapsack problem: a survey," *Naval Research Logistics Quarterly*, vol. 22, no. 1, pp. 127–144, 1975.

[6] L. Li, T. Flynn, and A. Hoisie, "Learning generalizable program and architecture representations for performance modeling," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '24, 2024. [Online]. Available: https://arxiv.org/abs/2310.16792

[7] J. Lee, S. Kim, S. Jang, J. Park, and Y. Kim, "Diffusion-based generative system surrogates for scalable learning-driven optimization in virtual playgrounds," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 9, no. 2, pp. 1–27, 2025.

[8] X. Ai, Z. Li, Y. Zhu, Z. Chen, S. Liu, and Y. Xu, "Netjit: Bridging the gap from traffic prediction to preknowledge for distributed machine learning," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 9, no. 1, pp. 1–25, 2025.

[9] H. Qi, L. Dai, W. Chen, Z. Jia, and X. Lu, "Performance characterization of large language models on high-speed interconnects," in *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 2023, pp. 53–60.

[10] T. Shi, Y. Yang, Y. Cheng, X. Gao, Z. Fang, and Y. Yang, "Alioth: A machine learning based interference-aware performance monitor for multi-tenancy applications in public cloud," *arXiv preprint arXiv:2307.08949*, 2023.

[11] D. Y. Yoon, Y. Wang, M. Yu, E. Huang, J. I. Jones, A. Kukkadapu, O. Kocas, J. Wiepert, K. Goenka, S. Chen, Y. Lin, Z. Huang, J. Kong, M. Chow, and C. Tang, "FBDetect: Catching tiny performance regressions at hyperscale through in-production monitoring," in *Proceedings of the 30th ACM Symposium on Operating Systems Principles*, ser. SOSP '24, 2024, pp. 522–540.

[12] Z. Liu, S. Zhang, and H. Zhao, "Survey of hardware acceleration of genomic analysis," in *2024 IEEE 17th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2024, pp. 540–547.

[13] X. Li, "Practical tools for reliable and reproducible graphics processing unit (gpu) computations," Ph.D. dissertation, The University of Utah, 2024.

[14] X. Zeng, K. Xie, Y. Li, K. Xu, G. Xie, J. Wen, Y. Cong, and W. Liang, "Int-mc: Low-overhead in-band network-wide telemetry based on matrix completion," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 8, no. 3, pp. 1–30, 2024.

[15] C. Fang, H. Liu, M. Miao, J. Ye, L. Wang, W. Zhang, D. Kang, B. Lyv, P. Cheng, and J. Chen, "Vtrace: Automatic diagnostic system for persistent packet loss in cloud-scale overlay network," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 31–43.

[16] M. Toslali, S. Qasim, S. Parthasarathy, F. Oliveira, H. Huang, G. Stringhini, Z. Liu, and A. Coskun, "An online probabilistic distributed tracing system," *arXiv preprint arXiv:2405.15645*, 2024.

[17] Uber Technologies, "CRISP: Critical path analysis for microservice performance," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '23, 2023.

[18] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi *et al.*, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 34–50.

[19] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[20] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 378–393.

[21] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic detection of performance deviations in the load testing of large scale systems," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 1012–1021.

[22] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A cost-aware logging mechanism for performance diagnosis," in *Proceedings of the 2015 USENIX Annual Technical Conference*, ser. USENIX ATC '15, 2015, pp. 139–150.

[23] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "Log20: Fully automated optimal placement of log printing statements under specified overhead threshold," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, 2017, pp. 565–581.

[24] Y. Zhang, T. Liu, Y. Wang, Y. Qi, K. Ji, J. Tang, X. Wang, X. Li, and Z. Zuo, "Hardtaint: Production-run dynamic taint analysis via selective hardware tracing," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1615–1640, 2024.

[25] S. Priyadarshan, H. Nguyen, R. Chouhan, and R. Sekar, "{SAFER}: Efficient and {Error-Tolerant} binary instrumentation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1451–1468.

[26] A. Eghbali and M. Pradel, "DynaPyt: A dynamic analysis framework for Python," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '22, 2022.

[27] B. Chen and Z. M. Jiang, "Characterizing logging practices in Java-based open source software projects—a replication study in Apache Software Foundation," *Empirical Software Engineering*, vol. 22, no. 1, pp. 330–374, 2017.

[28] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 102–112.

[29] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on Linux: Design, implementation, and overhead," *ACM Computing Surveys*, vol. 51, no. 2, pp. 26:1–26:33, 2018.

[30] J. Shen, H. Zhang, Y. Xiang, X. Shi, X. Li, Y. Shen, Z. Zhang, Y. Wu, X. Yin, J. Wang *et al.*, "Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 420–437.

[31] A. Nõu, A. Tarvo, and M. Dumas, "Investigating performance overhead of distributed tracing in microservices and serverless systems," in *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '24, 2024, pp. 162–166.

[32] I. Orton and A. Mycroft, "Tracing and its observer effect on concurrency," in *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2021, pp. 88–96.

[33] B. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX Annual Technical Conference, General Track*, 2004, pp. 15–28.

[34] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, vol. 20, no. 1, pp. 1–27, 2015.

[35] E. D. Berger, S. Stern, and J. A. Pizzorno, "Triangulating python performance issues with {SCALENE}," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 51–64.

[36] Z. Xu, Y. Chon, Y. Su, Z. Tan, S. Apostolakis, S. Campanoni, and D. I. August, "Prompt: A fast and extensible memory profiling framework," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 449–473, 2024.

[37] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage, "Be conservative: Enhancing failure diagnosis with proactive logging,"

in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '12, 2012, pp. 293–306. [Online]. Available: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-109.pdf

[38] M. M. Morovati, A. Bahrami, and V. Garousi, "Fault localization in deep learning-based software: A system-level approach," 2024, arXiv preprint arXiv:2411.08172.

[39] S. Ji, S. Lee, C. Lee, Y.-S. Han, and H. Im, "Impact of large language models of code on fault localization," in *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2025, pp. 302–313.

[40] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," in *OSDI*, vol. 4, 2004, pp. 16–16.

[41] B. Rahdari, P. Brunet, D. C. Petriu *et al.*, "Semantic models of performance indicators: A systematic survey," *ACM Computing Surveys*, 2024.

[42] P. Wilson, M. Davis *et al.*, "Region-based adaptive sampling for bounded overhead monitoring," in *Proceedings of the USENIX Annual Technical Conference*, ser. ATC '23, 2023.

[43] E. Ates, L. Sturmann, M. Toslali, O. Krieger, R. Megginson, A. K. Coskun, and R. R. Sambasivan, "An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 165–170.

[44] K. Yao, G. B. D. Padua, W. Shang, C. Sporea, A. Toma, and S. Sajedi, "Log4perf: Suggesting and updating logging locations for web-based systems' performance monitoring," *Empirical Software Engineering*, vol. 25, no. 1, pp. 488–531, 2020.

[45] M. Cinque, D. Cotroneo, A. Pecchia, and S. Russo, "A lightweight approach for injecting faults in the Linux TCP/IP protocol stack," in *Proceedings of the 2013 IEEE 6th International Conference on Software Testing, Verification and Validation*, 2013, pp. 264–273.

[46] R. Zhong, Y. Li, J. Kuang, W. Gu, Y. Huo, and M. R. Lyu, "Logupdater: Automated detection and repair of specific defects in logging statements," *ACM Transactions on Software Engineering and Methodology*, 2025.

[47] X. Wang, Z. Li, and Z. Ding, "Defects4log: Benchmarking llms for logging code defect detection and reasoning," *arXiv preprint arXiv:2508.11305*, 2025.

[48] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling," in *6th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '04, 2004.

[49] J. Shen, H. Zhang, Y. Xiang, X. Shi, X. Li, Y. Chen, X. Liu, Z. Song, Y. Liu, Z. Zhang, Z. Guo, J. Shen, and X. Wang, "Network-centric distributed tracing with DeepFlow: Troubleshooting your microservices in zero code," in *Proceedings of the ACM SIGCOMM 2023 Conference*, ser. SIGCOMM '23, 2023.

[50] S. Ashok, V. Harsh, B. Godfrey, R. Mittal, S. Parthasarathy, and L. Shwartz, "Traceweaver: Distributed request tracing for microservices without application modification," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 828–842.

[51] J. Chen, H. Li, W. Shang, and A. E. Hassan, "UniLog: Automatic logging via LLM and in-context learning," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24, 2024.

[52] X. Xie, Z. Cai, S. Chen, and J. Xuan, "Fastlog: An end-to-end method to efficiently generate and insert logging statements," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 26–37.

[53] Z. Ding, H. Li, W. Shang, and A. E. Hassan, "Go static: Contextualized logging statement generation," in *Proceedings of the 32nd ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '24, 2024.

[54] Y. Li, Y. Huo, Z. Jiang, R. Zhong, P. He, Y. Su, L. C. Briand, and M. R. Lyu, "Exploring the effectiveness of llms in automated logging statement generation: An empirical study," *IEEE Transactions on Software Engineering*, 2024.

[55] A. Mastropaolo, S. Scalabrino, N. Cooper *et al.*, "Log statements generation via deep learning: Widening the support provided to developers," *Journal of Systems and Software*, vol. 196, 2023.

[56] T. He, R. Rocha, and A. Jannesari, "CSI framework: Compiler-inserted instrumentation for race detection and deterministic replay," in *Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '21, 2021.

[57] D. Lehmann, J. Kinder, and M. Pradel, "Wastrumentation: Portable WebAssembly dynamic analysis with support for intercession," in *Proceedings of the 39th European Conference on Object-Oriented Programming*, ser. ECOOP '25, 2025.

[58] Z. Chen, Y. Zhu, and Z. Wang, "Design and implementation of an aspect-oriented c programming language," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 642–669, 2024.

[59] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "{ServerlessLLM}:{Low-Latency} serverless inference for large language models," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 135–153.

[60] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok, "Software monitoring with controllable overhead," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 327–347, 2012.

[61] M. Vierhauser, R. Wohlrab, M. Stadler, and J. Cleland-Huang, "Amon: A domain-specific language and framework for adaptive monitoring of cyber-physical systems." *J. Syst. Softw.*, vol. 195, p. 111507, 2023.

[62] B. Batoun, W. Shang, and A. E. Hassan, "Logging practices in software engineering: A systematic mapping study," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3168–3189, 2023.

[63] G. M. Tchamgoue and S. Fischmeister, "Lessons learned on assumptions and scalability with time-aware instrumentation," in *Proceedings of the 13th International Conference on Embedded Software*, 2016, pp. 1–7.

[64] A. Lyons, J. Gamba, A. Shawaga, J. Reardon, J. Tapiador, S. Egelman, N. Vallina-Rodriguez *et al.*, "Log: It's big, it's heavy, it's filled with personal data! measuring the logging of sensitive information in the android ecosystem," in *Usenix Security Symposium*, 2023.

[65] V. Horkỳ, J. Kotrč, P. Libič, and P. Tuma, "Analysis of overhead in dynamic java performance monitoring," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016, pp. 275–286.

[66] Z. Wang, T. Ma, L. Kong, Z. Wen, J. Li, Z. Song, Y. Lu, G. Chen, and W. Cao, "Zero overhead monitoring for cloud-native infrastructure using {RDMA}," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 639–654.

[67] L. Weng, P. Huang, J. Nieh, and J. Yang, "Argus: Debugging performance issues in modern desktop applications with annotated causal tracing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 193–207.

[68] D. M. Ogle, K. Schwan, and R. Snodgrass, "Application-dependent dynamic monitoring of distributed and parallel systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 7, pp. 762–778, 2002.

[69] J. Shastri, X. Wang, B. A. Shivakumar, F. Verbeek, and B. Ravindran, "Hmtrace: Hardware-assisted memory-tagging based dynamic data race detection," *arXiv preprint arXiv:2404.19139*, 2024.

[70] E. W. L. Leng, M. Zwolinski, and B. Halak, "Hardware performance counters for system reliability monitoring," in *2017 IEEE 2nd international verification and security workshop (IVSW)*. IEEE, 2017, pp. 76–81.

[71] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation," *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, pp. 1–12, 2010.

[72] A. S. Filho, R. J. Rodríguez, and E. L. Feitosa, "Evasion and countermeasures techniques to detect dynamic binary instrumentation frameworks," *Digital Threats: Research and Practice (DTRAP)*, vol. 3, no. 2, pp. 1–28, 2022.

[73] N. Kumar, B. R. Childers, and M. L. Soffa, "Low overhead program monitoring and profiling," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 1, pp. 28–34, 2005.

[74] T. Akinyemi, E. Solomon, A. Woubie, and K. Lippert, "A comprehensive review of static memory analysis," *IEEE Access*, 2024.

[75] J. Wei, G. Zhang, J. Chen, Y. Wang, W. Zheng, T. Sun, J. Wu, and J. Jiang, "Loggrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 452–468.

[76] S. Xie, J. Wang, M. Li, P. Chen, J. Xuan, and B. Li, "Tracepicker: Optimization-based trace sampling for microservice-based systems," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 1802–1823, 2025.

[77] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2014, pp. 129–140.

[78] J. Peeck, J. Schlatow, and R. Ernst, "Online latency monitoring of time-sensitive event chains in safety-critical applications," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 539–542.

[79] H. Zheng, C. Huang, X. Han, J. Zheng, X. Wang, C. Tian, W. Dou, and G. Chen, "μmon: Empowering microsecond-level network monitoring with wavelets," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 274–290.

[80] B. R. Calder, R. Brennan, J. Marcus, C. Malzone, and P. Canter, "High-precision, high-accuracy timekeeping in distributed survey systems," *The International Hydrographic Review*, 2008.

[81] C. D. Guerrero and M. A. Labrador, "Traceband: A fast, low overhead and accurate tool for available bandwidth estimation and monitoring," *Computer Networks*, vol. 54, no. 6, pp. 977–990, 2010.

[82] Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, "Clio: A hardware-software co-designed disaggregated memory system," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 417–433.

[83] A. Agarwal, Z. Liu, and S. Seshan, "{HeteroSketch}: Coordinating network-wide monitoring in heterogeneous and dynamic networks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 719–741.

[84] S. Karus, "Opportunity costs in free open-source software," in *IFIP International Conference on Open Source Systems*. Springer, 2019, pp. 139–150.

[85] H. Kashif, P. Arafa, and S. Fischmeister, "Instep: A static instrumentation framework for preserving extra-functional properties," in *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2013, pp. 257–266.

[86] K. Thomas and B. Hallbert, "Long-term instrumentation, information, and control systems (ii&c) modernization future vision and strategy," Idaho National Lab.(INL), Idaho Falls, ID (United States), Tech. Rep., 2013.

[87] D. Zhang, W. Li, and X. Xiong, "Overhead line preventive maintenance strategy based on condition monitoring and system reliability assessment," *IEEE Transactions on power systems*, vol. 29, no. 4, pp. 1839–1846, 2014.

[88] J. Bogatinovski and O. Kao, "Auto-logging: Ai-centred logging instrumentation," in *2023 IEEE/ACM 45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2023, pp. 95–100.

[89] N. Kavvadias, P. Neofotistos, S. Nikolaidis, C. Kosmatopoulos, and T. Laopoulos, "Measurements analysis of the software-related power consumption in microprocessors," *IEEE Transactions on Instrumentation and Measurement*, vol. 53, no. 4, pp. 1106–1112, 2004.

[90] S. Ryffel, T. Stathopoulos, D. McIntire, W. Kaiser, and L. Thiele, "Accurate energy attribution and accounting for multi-core systems," 2009.

[91] X. Yang, X. Ren, S. Yang, and J. McCann, "A novel temporal perturbation based privacy-preserving scheme for real-time monitoring systems," *Computer Networks*, vol. 88, pp. 72–88, 2015.

[92] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 195–211.

[93] C. van Sloun, V. Woeste, K. Wolsing, J. Pennekamp, K. Wehrle, A. Saillard, J. Bauer, E. Wagner, I. B. Fink, M. Schmidt *et al.*, "Detecting ransomware despite i/o overhead: A practical multi-staged approach," in *Network and Distributed System Security Symposium*, vol. 978, no. 3543-3545. Internet Society, 2025, pp. 3543–3545.

[94] K. Yao, G. B. de Pádua, W. Shang, S. Sporea, A. Toma, and S. Sajedi, "Log4perf: Suggesting logging locations for web-based systems' performance monitoring," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018, pp. 127–138.

[95] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 415–425.

[96] P. Arafa, G. M. Tchamgoue, H. Kashif, and S. Fischmeister, "Qdime: Qos-aware dynamic binary instrumentation," in *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2017, pp. 132–142.

[97] P. Arafa, H. Kashif, and S. Fischmeister, "Dime: time-aware dynamic binary instrumentation using rate-based resource allocation," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. IEEE, 2013, pp. 1–10.

[98] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, pp. 105–118, 2005.

[99] S. Zhang and M. D. Ernst, "Proactive detection of inadequate diagnostic messages for software configuration errors," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 12–23.

[100] ——, "Which configuration option should i change?" in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 152–163.

[101] D. Maplesden, K. Von Randow, E. Tempero, J. Hosking, and J. Grundy, "Performance analysis using subsuming methods: An industrial case study," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 149–158.

[102] D. Maplesden, E. Tempero, J. Hosking, and J. C. Grundy, "Subsuming methods: Finding new optimisation opportunities in object-oriented software," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, pp. 175–186.

[103] W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using regression models on clustered performance counters," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015, pp. 15–26.

[104] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *2009 IEEE International Conference on Software Maintenance*. IEEE, 2009, pp. 125–134.

[105] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 24–33.

[106] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou, "The game of twenty questions: Do you know where to log?" in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 125–131.

[107] C. Shannon, "A mathematical theory of communication. the bell systems technical journal, 27: July 379–423," 1948.

[108] Y. Zhang, X. Chang, L. Fang, and Y. Lu, "Deeplog: Deep-learning-based log recommendation," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2023, pp. 88–92.

[109] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: Assessment of a critical software development process," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 169–178.

[110] O. Crameri, R. Bianchini, and W. Zwaenepoel, "Striking a new balance between program instrumentation and debugging time," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 199–214.

[111] T. Jia, Y. Li, C. Zhang, W. Xia, J. Jiang, and Y. Liu, "Machine deserves better logging: a log enhancement approach for automatic fault diagnosis," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 106–111.

[112] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems," in *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[113] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling." in *OSDI*, vol. 4, 2004, pp. 18–18.

[114] H. Lucas Jr, "Performance evaluation and monitoring," *ACM Computing Surveys (CSUR)*, vol. 3, no. 3, pp. 79–91, 1971.

[115] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[116] B. Chen and Z. M. Jiang, "A survey of software log instrumentation," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–34, 2021.

[117] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards automated log parsing for large-scale log data analysis," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 6, pp. 931–944, 2017.

[118] R. Zhou, M. Hamdaqa, H. Cai, and A. Hamou-Lhadj, "Mobilogleak: A preliminary study on data leakage caused by poor logging practices," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 577–581.

[119] T. Xiao, Z. Quan, Z.-J. Wang, K. Zhao, and X. Liao, "Lpv: A log parser based on vectorization for offline and online log parsing," in *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2020, pp. 1346–1351.

[120] S. Gholamian and P. A. Ward, "A comprehensive survey of logging in software: From logging statements automation to log mining and analysis," *arXiv preprint arXiv:2110.12489*, 2021.

[121] D. Pisinger and P. Toth, "Knapsack problems," *Handbook of Combinatorial Optimization: Volume1–3*, pp. 299–428, 1998.

[122] H. Kellerer, U. Pferschy, and D. Pisinger, "Multidimensional knapsack problems," in *Knapsack problems*. Springer, 2004, pp. 235–283.

[123] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[124] C. Laaber, M. Basmaci, and P. Salza, "Predicting unstable software benchmarks using static source code features," *Empirical Software Engineering*, vol. 26, no. 6, p. 114, 2021.

[125] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, and H. Haghighi, "Sldeep: Statement-level software defect prediction using deep-learning model on static code features," *Expert Systems with Applications*, vol. 147, p. 113156, 2020.

[126] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, and C. Soubervielle-Montalvo, "Source code metrics: A systematic mapping study," *Journal of Systems and Software*, vol. 128, pp. 164–197, 2017.

[127] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study." *CLOSER (1)*, pp. 137–146, 2016.

[128] L. Wang, N. Zhao, J. Chen, P. Li, W. Zhang, and K. Sui, "Root-cause metric location for microservice systems via log anomaly detection," in *2020 IEEE international conference on web services (ICWS)*. IEEE, 2020, pp. 142–150.

[129] S. Gholamian and P. A. Ward, "Borrowing from similar code: A deep learning nlp-based approach for log statement automation," *arXiv preprint arXiv:2112.01259*, 2021.

[130] Y. Zhang, C. Ge, S. Hong, R. Tian, C. Dong, and J. Liu, "Delesmell: code smell detection based on deep learning and latent semantic analysis," *Knowledge-Based Systems*, vol. 255, p. 109737, 2022.

[131] Z. Li, T.-H. Chen, and W. Shang, "Where shall we log? studying and suggesting logging locations in code blocks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 361–372.

[132] S. Locke, H. Li, T.-H. P. Chen, W. Shang, and W. Liu, "Logassist: Assisting log analysis through log summarization," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3227–3241, 2021.

[133] P. C. Brebner, "Automatic performance modelling from application performance management (apm) data: an experience report," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016, pp. 55–61.

[134] P. Xiong, C. Pu, X. Zhu, and R. Griffith, "vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, 2013, pp. 271–282.

[135] R. Gao, Z. M. Jiang, C. Barna, and M. Litoiu, "A framework to evaluate the effectiveness of different load testing analysis techniques," in *2016 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 2016, pp. 22–32.

[136] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 9, pp. 2114–2129, 2019.

[137] R. Heinrich, A. Van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance engineering for microservices: research challenges and directions," in *Proceedings of the 8th ACM/SPEC on international conference on performance engineering companion*, 2017, pp. 223–226.

[138] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019, pp. 25–32.

[139] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, "The power of prediction: Microservice auto scaling via workload learning," in *Proceedings of the ACM Symposium on Cloud Computing*, 2022.

[140] I. Cohen, Y. Huang, J. Chen, J. Benesty, J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," *Noise reduction in speech processing*, pp. 1–4, 2009.

[141] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in neural information processing systems*, vol. 30, 2017.

[142] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[143] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[144] X.-w. Chen and J. C. Jeong, "Enhanced recursive feature elimination," in *Sixth international conference on machine learning and applications (ICMLA 2007)*. IEEE, 2007, pp. 429–435.

[145] C. Reeves and J. E. Rowe, *Genetic algorithms: principles and perspectives: a guide to GA theory*. Springer Science & Business Media, 2002, vol. 20.

[146] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 24–33.

[147] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards just-in-time suggestions for log changes," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1831–1865, 2017.

[148] H. Li, T.-H. P. Chen, W. Shang, and A. E. Hassan, "Studying software logging using topic models," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2655–2694, 2018.

[149] H. Li, W. Shang, and A. E. Hassan, "Where shall we log? studying and suggesting logging locations in code blocks," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 361–372.

[150] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Deeplv: suggesting log levels using ordinal based neural networks," in *Proceedings of the 43rd International Conference on Software Engineering*. IEEE Press, 2021, pp. 1200–1212.

[151] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu, "Which variables should i log?" *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1778–1792, 2019.

[152] Y. Zeng, J. Chen, W. Shang, and T.-H. Chen, "Studying the characteristics of logging practices in mobile apps: a case study on f-droid," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3394–3434, 2019.

[153] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Studying the relationship between logging characteristics and the code quality of platform software," *Empirical Software Engineering*, vol. 20, no. 1, pp. 1–27, 2015.

[154] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. ACM, 2011, pp. 3–14.

[155] S. Gholamian, "Leveraging natural language processing and code clones to predict log placement and content," *arXiv preprint arXiv:2108.12837*, 2021.

[156] H. Li, W. Shang, and A. E. Hassan, "Dlfinder: characterizing and detecting duplicate logging code smells," *Proceedings of the 42nd International Conference on Software Engineering*, pp. 152–163, 2020.

[157] Z. Ding, H. Li, W. Shang, and A. E. Hassan, "Logentext: fully automated logging statement generation using neural machine translation," in *Proceedings of the 44th International Conference on Software Engineering*. ACM, 2022, pp. 2267–2278.

[158] J. Chen, H. Li, W. Shang, and A. E. Hassan, "Unilog: automatic logging via llm and in-context learning," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, 2024, pp. 1–13.

[159] B. Zhang, H. Li, W. Shang, and A. E. Hassan, "Studying logging practice in test code," *Empirical Software Engineering*, vol. 27, no. 4, pp. 1–44, 2022.

[160] H. Li, Z. Xu, W. Shang, and A. E. Hassan, "Are they all good? studying practitioners' expectations on the readability of log messages," pp. 1456–1468, 2023.

[161] B. Batoun, W. Shang, and A. E. Hassan, "A literature review and existing challenges on software logging practices," *Empirical Software Engineering*, vol. 29, no. 3, pp. 1–58, 2024.

[162] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering," *ACM computing surveys (CSUR)*, vol. 54, no. 6, pp. 1–37, 2021.